# アセンブラ入門 CASLⅡ

神奈川大学工学部 助教授
内 田 智 史 著

アセンブラシミュレータ収録
CD-ROM付属
（Windows版）

電子開発学園出版局


COMPACT disc
アセンブラ入門（CASL Ⅱ）
付属 CD-ROM
Windows 専用
■このCD-ROMは本書をご覧になった上でご使用ください
■このCD-ROMを使用したことによって何らかのトラブルが生じたとしても、出版社及び著作権者は一切の責任を負いません
収録内容
●CASL シミュレータ CASLDV（Ver.1.01）
●マニュアル（PDFファイル）
●ソースプログラム（本書掲載プログラム）
発行　電子開発学園出版局
発売　株式会社SCC
※MS、Microsoft、Windows、MS-DOSは米国のMicrosoft Corporation の米国およびその他の国における登録商標です
※その他、記載されている会社名、製品名等は各社の商標または登録商標です
© Satoshi Uchida 2002 All Rights Reserved

情報処理基礎講座

# アセンブラ入門(CASL Ⅱ)

神奈川大学工学部 助教授　内 田 智 史 著

電子開発学園出版局

# 序　　　文

COMET IIは、経済産業省の情報処理技術者試験の基本情報技術者試験におけるアセンブラ言語の問題を出題するために設計されたコンピュータです。試験のために設計されているので、実在するコンピュータに比べてとても単純な構成になっています。COMET IIのハードウェアは、実際には誰も作ったことがないので、仮想的なコンピュータと呼ばれることもあります。

CASL IIは、COMET IIに対するアセンブラ言語です。アセンブラ言語とは、そのコンピュータの機械語に1対1に対応して作られた言語です。

情報処理技術者試験は、昭和44年(1969年)に試験制度が発足しました。これにより、 第一種情報処理技術者認定試験と第二種情報処理技術者認定試験が実施されました。COMET・CASLは、第一種情報処理技術者認定試験では必須項目、第二種情報処理技術者認定試験では選択項目となっていました。平成6年(1994年)に、新しい試験制度へ移行しましたが、第一種情報処理技術者認定試験と第二種情報処理技術者認定試験はそのまま残りました。しかし、この時の改訂で、第一種情報処理技術者試験からプログラミング能力を問う問題がなくなり、実質的にCOMET・CASLは第二種情報処理技術者試験のみで出題されるようになりました。

平成13年(2001年)に、さらに試験制度が変更になり、第一種情報処理技術者認定試験がソフトウェア開発技術者試験に、第二種情報処理技術者認定試験が基本情報技術者試験に代わり、COMET・CASLも新しく改訂され、COMET II・CASL IIという名称になりました。本書では、このCOMET II・CASL IIを対象としています。

本書を書く際にあたって、最初から読み進んで一度読めば、必ず分かるように心がけました。しかし、プログラミングの入門書である以上、CASL IIのリファレンスマニュアル的な部分も必要なため、COMET IIの機械語の詳細な説明を第６章にまとめてあります。この部分は、プログラミングの概念と密接に絡むため、最初に読んだ段階では、ピンと来ない部分も多いと思います。したがって、第６章は，最初は、流し読みをして、第７章以降を読みながら必要に応じて読み戻るというのがよろしいかと思います。

また、このようなプログラミング言語の習得では、実際に、プログラムを作成してコンピュータにかけて実行するという訓練が必要になってきます。そこで、本書では、Windows (Windows95, Windows98, Windows 2000)をお持ちの読者に対しては、筆者が作成したCASL IIの開発環境CASLDV(Ver.1.0) を付属のCD-ROMに用意して、読者が学習しやすいように配慮しました。これは、エディタやデバッガを含む本格的なもので、読者が作成したCASL IIのプログラムがCOMET II上で動作する様子をグラフィックスでよく分かるようにしました。また、CD-ROMの中にはサンプルプログラムを入れてありますので、本書を学習しながら、ぜひ、サンプルプログラムも実行させてみてください。なお、Unix版のシミュレータ(ただし、コマンドライン版のみ)をご希望の方は、筆者の研究室(神奈川大学工学部経営工学科情報システム工学研究室)のWebページ(http://www.inf.ie.kanagawa-u.ac.jp/)

をご覧下さい。

また、CASLDVは単にプログラムを組むことを支援するだけでなく、COMETⅡの機械語命令の動作の様子をCPUや主記憶装置との関係とともにグラフィカルに示してくれるので、初心者が機械語の動作のイメージを学習するにはたいへん適していると考えています。初心者の方には、CASLDVのこの表示機能を活用していただき、アセンブラ言語の理解を深めていただきたいと思います。

CASLDVのマニュアル(約50ページ)はCD-ROMの中にPDFファイルとして納められています。このPDFファイルを読むにはAcrobat Readerが必要ですが、このソフトウェアもCD-ROMの中に納められています。なお、CASLDVの簡易マニュアルを次ページの「付属CD-ROMについて」として記載しています。

アセンブラ言語を勉強すると、他のプログラミング言語、たとえばC言語やJavaなどの理解が、さらに一歩深まります。その意味では、多くの方にCASL IIを学習していただきたいのですが、実際問題としては、試験対策のために本書を読まれる方も多いと思われます。そこで、本書の中にはできるだけ多くの演習問題を取り入れました。第9章までは比較的簡単な演習問題を取り入れてあります。これらを解いて、問題を解くコツをつかんでください。第10章からは、実際の試験問題に近い本格的な問題を数多く取り入れてあります。できるだけ、独力でこれらの問題をたくさん解いて力をつけてください。第13章には、過去の実際の問題を解く方法について解説してあります。

本書を執筆するにあたり、SCC出版局の方々、特に久富義昭氏、関明浩氏、前田博幸氏には、遅れがちな原稿を辛抱強く待っていただき、本当に感謝しております。また、本書の作成に携わった多くの方々に感謝いたします。特に本書の草稿をお読みいただき、さまざまなご指摘をいただいた電子開発学園の先生方に感謝致します。さらに、本書に付属しているCASL II開発環境CASLDVの動作確認などをしていただいた方々に感謝したいと思います。

2001年2月

内田智史

# ●付属CD-ROMについて

本書に付属のCD-ROMは、Windows(95以上) に対応しています。CD-ROMにはアセンブラシミュレータ「CASLDV(Ver.1.01)」とそのマニュアル、マニュアルを読むための「Acrobat Reader」が収録されています。なお、「Acrobat Reader」のインストール方法についてはCD-ROM内で説明しています。

## ★CASLDVのインストールとアンインストール★

CASLDVのインストール方法は、本書に付属のCD-ROMのCASLDVマニュアルにも詳しく記載されていますが、ここでも簡単に説明することにします。

CASLDVをインストールするには付属のCD-ROMのInstallフォルダの中のSetup.exeを実行してください。すると、インストーラが起動します。後は、インストーラの指示に従ってインストールしてください。その中で製品ライセンス契約の部分がありますのでその内容に同意される方だけご使用ください。また、インストールするフォルダを聞いてくるので指定してください。指定したフォルダの中に、tempとbinというフォルダが作成され、CASLDVの本体CASLDV.exeがbinの中にコピーされます。このショートカットをディスクトップ上に作ると便利です。

CASLDVをアンインストールするには、「スタート」メニューから「設定」を選び、「コントロールパネル」を表示し、「アプリケーションの追加と削除」を実行し、CASLDVを選んでください。なお、場合によって作成したフォルダが全て削除できないこともあります。そのときは手作業で削除してください。

## ★CASLDVの実行★

まず、CASL IIのプログラムを格納するフォルダを作成してください。CASLDVを起動するには、「スタート」メニューから「プログラム」を選択し、その中から「CASLシミュレータ」→「CASLシミュレータ」を選んでください。ショートカットがある場合には、それをダブルクリックするかそこにファイルをドラッグすると起動します。

起動すると、左上・右上・下の部分に3つのペイン(画面)が表示されます。これをアセンブラリストペイン(左上)、エディタペイン(右上)、インフォメーションペイン(下)と呼びます。エディタペインにCASL IIのプログラムを入力してください。ファイル名は「~.casl」として、ディスク装置などに保存してください。

「アセンブル」メニューから、「アセンブル(CASL完全準拠)」を選び、アセンブラを起動してください。エラーがある場合、インフォメーションペインにエラーメッセージが表示されるので、それをダブルクリックしてください。エディタの該当する行が反転し、位置を知らせてくれます。エラーがなくなると、アセンブラリストペインにアセンブラリストが表示されます。なお、アセンブラの起動は、ツールバー上のボタンでも行えます。

プログラムを実行させるには、「実行」メニューから「実行開始」を選んでください。インフォメーションペインにトレース結果が表示されます。なお、「実行開始」もツールバー上に対応するボタンがあります。

## ★CD-ROMの内容★

本書に付属のCD-ROMには、ソフトウェア(CASLDV)とそのマニュアルだけでなく、本書で提供しているすべてのソースプログラムと演習問題で出題されたすべてのソースプログラムが含まれています。Sourceフォルダの中には各章に対応したフォルダがあり、その中に本書で述べられているプログラムのソースファイルがあります。Answerフォルダの中にも各章に対応したフォルダがあり、その中に演習問題のプログラムがあります。「高度なサンプル」というフォルダには、本書では言及していないが読者のためになる高度なレベルのプログラムが入っています。そこには、HANOI.casl(ハノイの塔)、PRIME.casl(素数)、SORT.casl(ソート、並び替え)のプログラムがありますので、参考にしてください。

## ★CASLDVにおけるプログラムの実行★

CASLDVでは、次の方法でプログラムの実行ができます。

(1) 高速実行
(2) トレース実行
(3) 1命令ずつのステップ実行
(4) ブレークポイントまでの実行

(1)と(2)はトレースを行うか行わないかの違いだけで実質的な違いはありませんが、実行速度がかなり異なります。トレースの指定は、「オプション」メニューから「実行時オプションの指定」を選び、そのダイアログの「さまざまな実行時オプション」グループの中の「実行時とレースを出力する」で行います。これがチェックされていれば実行時とレースを行い(規定値)、チェックされていなければ行いません。いずれにせよ、(1)、(2)は「実行」メニューの「実行開始」によって行います。なお、このトレース機能は、次で説明する(3)、(4)でも有効です。

(3)と(4)は実行の様子をグラフィカルに示すモードです。このモードでは、まず、「実行」メニューの「ステップ実行開始」を選択します。この後で、↓(矢印)キーを押すと1命令分ステップ実行します。ステップモードの時に、「実行」メニューから「ブレークポイントまで実行」を選ぶと設定されたブレークポイントまで一気に実行します。ブレークポイントを設定するには、アセンブラリストペインに示されたアセンブラリストの機械語のアドレスをマウスでクリックします。解除するには、そこを再びクリックします。なお、ブレークポイントはいくつでも設定できます。

(3)と(4)のモードでは、プログラムの実行の様子がグラフィカルに示されるため、実行のメカニズムをより深く理解できるようになると思います。

なお、実行に関するメニューは、ツールバー中のボタンにもありますので、実際にはそちらをお使いください。

## ★CASLDVにおけるさまざまな表示機能★

CASLDVには、読者がCASL IIのプログラミングを効率よく学習できるように、さまざま表示機能が用意されています。それらの表示機能を使うには、メニューの中から「表示」を選択します。

「OUT命令の出力表示」を選択すると、OUT命令によって出力されるデータを表示するダイアログが表示されます。OUT命令による出力は、インフォメーションペインにも出力されますが、他のトレ

ース情報の中に埋没することが多いので、このダイアログが用意されています。
「レジスタの表示」は、汎用レジスタの値を、さまざまな表現方法で表示します。ここに表示されているレジスタの値は、プログラムの実行と共にリアルタイムに変化します。
「記憶領域の表示」は、主記憶装置の中を表示する機能ですが、ラベルで指定したり、直接アドレスで指定することができます。
「メモリマップとロード状況」は、COMET IIの主記憶装置の概観を示し、プログラムのロード場所やスタックの位置などを表示します。

# 目　次

# アセンブラ入門(CASL Ⅱ)

# 第1章 COMET IIの概要とその位置付け

## 1.1 COMET IIとCASL IIの概要

### 1.1.1 はじめに

これから私たちは、COMET II（コメットツー）というコンピュータを動かすためのCASL II（キャッスルツー）という言語を学習していきます。そのためには、多少なりとも、コンピュータについての理解が必要になります。コンピュータというと今では誰もがパーソナルコンピュータを思い浮かべるでしょう。でも、パーソナルコンピュータとCOMET IIの間には、とてつもなく大きなギャップが存在します。このギャップは、コンピュータのことを良く知らないとなかなか理解できません。そこで、まず最初に、コンピュータがどのように発明され、発展していったかについて簡単に説明したいと思います。この中で、コンピュータの動作原理、機械語やアセンブラの役割について説明します。

なお、COMET IIは、情報処理技術者試験のために考え出されたコンピュータであり、現実には存在しません。COMET IIの詳細な定義は、情報処理技術者試験案内の中にある「仕様」に記載されています。なお、この「仕様」は本書の最後にも記載されていますが、情報処理技術者試験の問題の中にも記載されています。

### 1.1.2 初期のコンピュータ

計算を補助する装置として、最近ではあまり見かけませんが、算盤（そろばん）があります。このような計算に関する装置には、算盤を始め、歯車を用いた機械的な装置が数多く発明されています。しかし、コンピュータは電子の力を利用しています。電子式の最初のコンピュータは、1937年から1941年頃に開発されたジョン・V・アタナソフとクリフォード・ベリーの**アタナソフ・ベリー・コンピュータ**であると言われています(米国アイオワ州立大学)(*1)。これは、線形連立方程式を解くためのコンピュータで、2進法の採用や論理演算回路という現在のコンピュータでは常識となっている基本原理を確立しました。そののち、1942年から1946年にかけて、非常に有名な**ENIAC**(Electronic Numerical Integrator And Computer)という世界で最初に実用化されたコンピュータを、米国ペンシルベニア大学ムーア電気工学科のジョン・モークリーとプレスパー・エッカートが開発し、それ以降の輝かしいコンピュータの発展が始まったのです。ENIACは18,000本の真空管を使ったもので、30トンの重量があり、非常にメンテナンスが大変でした。また、ENIACに計算を行わせるためには、その手順をENIACに指示しなければなりません。この指示は、パッチボードと呼ばれる特殊な配線盤の上に配線を行うことによって行っていました。計算手順を変更するには、パッチボードの配線を変更しなければならず、大変な作業だったようです。

(*1) アタナソフについては、クラーク・R・モレンホフ著、最相力/松本泰男共訳、「ENIAC神話の崩れた日」、工業調査会、に詳しく載っています。

### 1.1.3 プログラムの登場とプログラム内蔵方式

そのころ、フォン・ノイマンという学者が、コンピュータに対して計算手順を示すには、配線に頼るのではなくて、ちょうど我々が文章を書くように、計算手順を示した命令書を作成して、それをコンピュータの主記憶装置の中に計算するデータとともに格納し、コンピュータは、その命令書に書かれた命令文を主記憶装置から逐一取り出して、それを実行するべきであるという提案を行いました。この計算手順を示した命令書のことを**プログラム**と呼びます。フォン・ノイマンのこの考え方は、プログラムとデータを区別なく主記憶装置に格納するという点が優れており、**プログラム内蔵方式**と呼ばれ、現在のほとんどすべてのコンピュータの基本原理になっています。

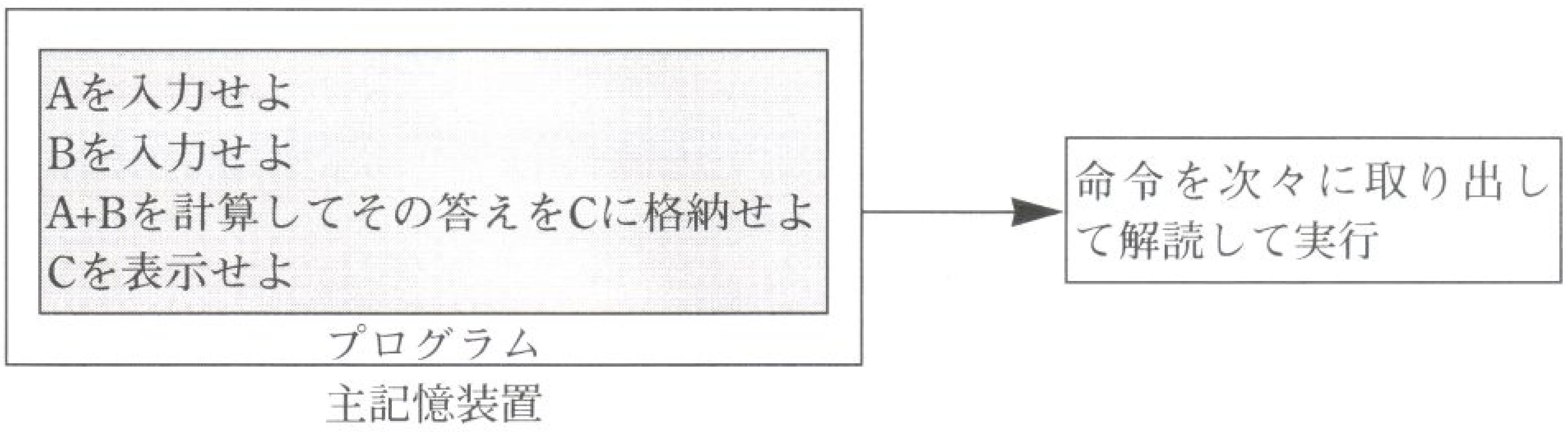



**図1-1 プログラム内蔵方式**

プログラム内蔵方式を初めてコンピュータに取り入れたのは、イギリス・ケンブリッジ大学のモーリス・ウィルスによる**EDSAC**(Electronic Delay Storage Automatic Calculator, 1949年稼動)や、米国ペンシルベニア大学の**EDVAC**(Electronic Discrete Variable Automatic Computer, ENIACの後継機、1951年)というコンピュータです。

その後、商用のためのコンピュータ Univac Iがレミントン・ランド(後にスペリー・ランド社、さらに現在はユニシス)によって開発され1951年にその一号機が設置されました。これは、真空管を5600本使い、計算は2進法（第２章参照）ではなく10進法で12桁まで計算できました。さらに、1952年にIBMがIBM701を開発し、これ以降コンピュータの商用利用が爆発的に増大しました。

### 1.1.4 機械語の意味

EDSACやEDVACは、基本的には現在のコンピュータとほぼ同じ原理で動作します。コンピュータを実行させるには、**機械語**と呼ばれる言語で書かれた命令書を用意します。これがプログラムと呼ばれるものです。コンピュータは、プログラムの中に書かれた機械語を淡々と実行する単なる機械なのです。

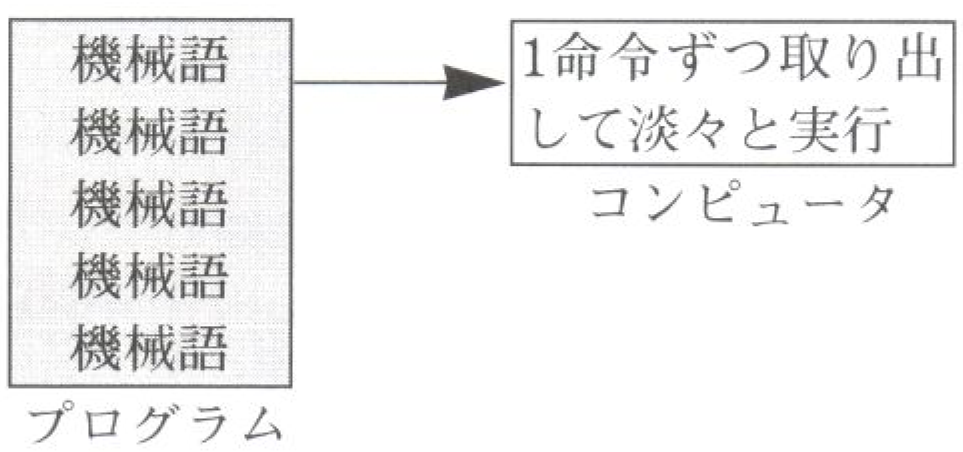



**図1-2 機械語とその実行**

さて、機械語は、コンピュータごとに異なっています。あるコンピュータのために書かれた機械語は、種類の違う別のコンピュータでは動作しません。EDVACの機械語は、EDVACというコンピュータのために用意された機械語です。EDVACの機械語のプログラムは、別のコンピュータ上では動作

しません。

これと同様に、COMET IIの機械語は、COMET IIというコンピュータのために用意された機械語です。COMET IIの機械語は、別のコンピュータ上ではそのままでは動作しません。

### 1.1.5 アセンブラ言語の登場

機械語によるプログラム開発は人間にとってとても困難です。次に示すものは、COMET IIの機械語(＊1)ですが、これが何であるか分かりますか。

```
0010010000010010
0010010100010011
0101000000010000
```

コンピュータであるCOMET IIは、これを正しく解釈し実行できます。しかし、人間にはそんなことはできません。そこで、この機械語を人間が直接記述するのではなく、この機械語を人間の分かる言葉に1対1に置き換えた言語が開発されました。これを**アセンブラ言語**(＊2)と呼びます。先ほどの機械語の隣にアセンブラ言語を添え、さらにその横に日本語による説明文を置きます。

| 機械語 | アセンブラ言語 | 日本語による説明 |
|---|---|---|
| 0010010000010010 | ADDA GR1,GR2 | GR1にGR2の値を加えよ |
| 0010010100010011 | SUBA GR1,GR3 | GR1からGR3の値を引け |
| 0101000000010000 | SLA GR1,2 | GR1の値を4倍せよ |

まだ、GR1やGR2が何であるのかを説明していないので、「日本語による説明」の意味がよく分からないと思いますが、「加算して、減算して、4倍している」ということくらいはわかるでしょう。また、アセンブラ言語の部分を読んでも多少は意味が分かるかと思います(しかし、そうは言ってもADDは「加える」、SUBは「引く」ということを意味しているというのは直ちに英語から分かりますが、SLAが4倍というのはCASL IIアセンブラ言語を学習しないと分からないですね)。前述のように、COMET IIの機械語に対応して作られたアセンブラ言語を**CASL II**といいます。

アセンブラ言語の登場により、機械語で直接プログラムを作成しなくても良いので、人間にとっては大助かりです。しかし、コンピュータは、アセンブラ言語を理解することはもちろんできません。そこで、アセンブラ言語で書かれたプログラムを実行するために、このアセンブラ言語で書かれたプログラムを機械語に翻訳する必要があります。この翻訳を人間が行っていたのでは大変ですから、これもまたコンピュータに任せます。この翻訳作業はアセンブラというプログラムを用いて行います。**アセンブラ**は、アセンブリ言語を入力してそれを機械語に変換する翻訳プログラムです。

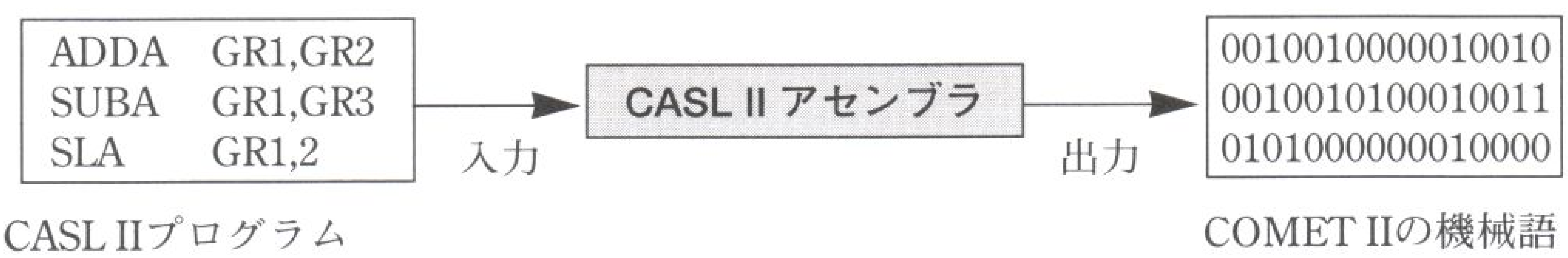



**図1-3 CASL II アセンブラとその役割**

CASL IIで書かれたプログラムを入力して、COMET IIの機械語プログラムを出力するアセンブラを**CASL IIアセンブラ**と呼びます。

(＊1) 本書では、COMET IIの機械語は情報処理技術者試験案内の中の参考の部分に示されているものに準拠しています。
(＊2) Assembly languageが英語なので、本来はアセンブリ言語とすべきですが、JISおよびそれに準拠している情報処理技術者試験では、アセンブラ言語としているので、本書でもアセンブラ言語と表現します。

前述のように、機械語はコンピュータによって異なります。たとえば、現在のパーソナルコンピュータで主流のPentiumの元祖である8086[(*1)]上のアセンブラ言語(CP/M-86)の場合には、次のようになります。

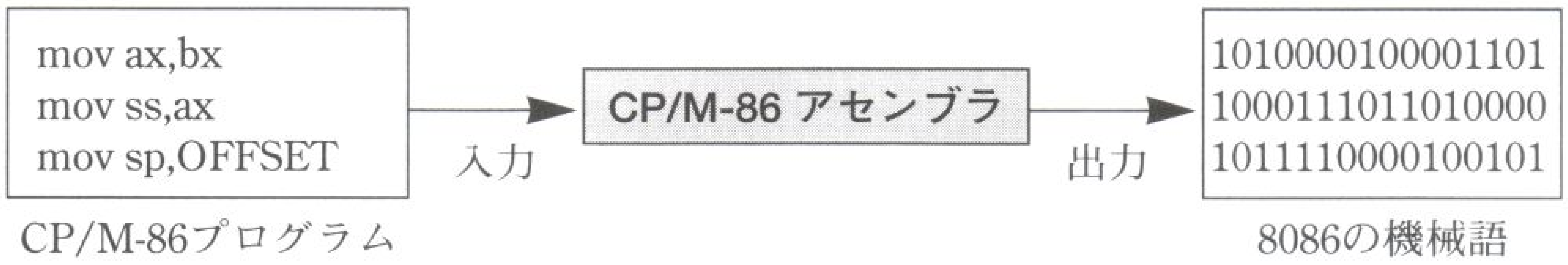



**図1-4　CP/M-86アセンブラとその役割**

CP/M-86では、小文字の記述を許していますが、CASL IIでは小文字の記述が許されていないので、プログラムは全て大文字で書くことになっています。

このように、コンピュータが異なれば、機械語が異なるので、それに応じてアセンブラ言語も異なります。

### 1.1.6　高水準言語とコンパイラの登場

さて、話をコンピュータの歴史に戻しましょう。1950年代も半ばになってくると、コンピュータの需要はますます増大し、プログラムを作成する効率が重視されるようになりました。たとえば、1950年代前半では科学技術系の研究者が、自分の研究のためにコンピュータを使うには、アセンブラ言語を学習し、さらには、複雑なコンピュータの動作原理を学習する必要があったのです。しかし、FORTRANという言語とその言語によって書かれたプログラムを機械語へ変換する仕組みが1950年代中ごろに開発され、人間にとって、とても理解しやすい形式でプログラムを作成できるようになったのです。FORTRANでは、プログラムの中に数式を書けば、それがそのまま機械語に変換されます。

人間の考え方にどのくらい近いのかという意味で当時は、アセンブラ言語のことを**低水準言語**、FORTRANのような言語のことを**高水準言語**と呼んでいました(現在でも、このように呼ぶことがあります)。高水準言語を機械語に翻訳するプログラムのことを**コンパイラ**と呼び、FORTRANプログラムを対応するコンピュータの機械語に翻訳するプログラムのことを**FORTRANコンパイラ**と呼びます。

FORTRANコンパイラのイメージを次に示します。




**図1-5　FORTRANコンパイラとその役割**

高水準言語は、その後、次々に開発され、COBOL(1960年)、PL/I(1965年)、C(1972年)、などが有名で情報処理技術者試験にも取り入れられていました(FORTRANとPL/Iは、現在では、試験からは除外されています)。

高水準言語が、アセンブラ言語と異なる点はどこでしょうか。それは、アセンブラ言語が機械語と1対1対応であるのに対して、高級言語は必ずしも1対1には対応していないという点が、まず挙げられます。基本的にアセンブラ言語では、機械語の命令一つ一つに対して、それに相当する命令がありま

(＊1) 8086というのはCPU（1.3節参照）の名前です。8086→80186→80286→80386→80486→Pentiumと進化してきました。

す。高級言語の場合、多くの場合、一つの命令が複数の機械語命令に変換されます。

次のプログラムは1～Nまでの和を求める問題です。それには、次の公式を使えば良いでしょう。

$$S = \sum_{k=1}^{N} k = \frac{1}{2} N \times (N + 1)$$

CASL IIとC言語で、このプログラムを作成すると、次のようになります。

```
SUMUPN START
       LD   GR1,N     1からNまでの和を求める
       LAD  GR7,1
       LD   GR2,GR1
       ADDA GR2,GR7   GR2←N+1
       XOR  GR3,GR3
LOOP   SUBA GR1,GR7   GR1のカウントダウン
       JMI  EXIT
       ADDA GR3,GR2
       JUMP LOOP
EXIT   SRA  GR3,1     GR3←GR3÷2
       ST   GR3,S
       RET
N      DC   100
S      DS   1
       END
```

CASL IIによるプログラム

```
#include<stdio.h>

int main(void)
{
     int S, N=100;

     S = N * (N+1) / 2;
     printf("S=%d¥n", S);
     return 0;
}
```

C言語によるプログラム

C言語によるプログラムでは、計算式が「`S=N*(N+1)/2;`」と書かれているように1行ですんでいるのに対し、CASL IIによるプログラムでは、その部分の計算式が9行にも及んでいます。つまり、C言語で書かれたプログラムは、Cコンパイラによって複数の機械語に展開されているのです。

### 1.1.7 最適化コンパイラの登場

FORTRANは、1950年代半ばに実用的なコンパイラが登場しました。当時のFORTRANコンパイラの開発目標の一つが、生成された機械語プログラムの実行効率の良さでした。当時、研究用のコンパイラは数多く開発されていました。しかし、一般に、コンパイラが生成する機械語は、優秀な人間が必死になって書いたプログラムに比べて、効率が格段に悪いので、FORTRANのようなプログラミング言語は実用的ではないと考えられていました。そこで、FORTRANコンパイラが生成する機械語は、人間が手で書いた機械語(実際にはアセンブラ言語で書いたプログラム)と同等の実行効率を出せるように工夫されたのです。

その後、さまざまな研究が行われ、その成果はコンパイラの最適化機能として集約され、現在では、その技術はかなりのレベルに達しています。**最適化機能**とは、プログラムを機械語に変換する際に、極力効率的になるように処理するだけでなく、場合によっては人間の書いたプログラムの実行手順まで変えてしまう機能のことです。実際に、現在のFortran(FORTRANは、その後数回の改定を経て、現在は1995年にJIS（日本工業規格）として定められたFortran95になっています)やC言語コンパイラの最適化技術は非常に優れています。

## 1.2 CASL IIアセンブラ言語を学習する価値

さて、それでは、私たちがCASL II アセンブラ言語を学習する価値はどこにあるのでしょうか。最適化コンパイラの機能が優れていて、人間がアセンブラ言語で書いたプログラムよりも優れた機械語を生成するのであれば、アセンブラ言語でプログラムを書く必要がないではないかと思う人も多いでしょう。

筆者は次のように考えます。

### (1) 現在でも、アセンブラ言語で書かざるを得ない部分がある

いくら最適化コンパイラが優れているといっても、何らかのプログラミング言語を使う以上、その言語の制約を受けます。たとえば、C言語を使う場合には、C言語の持つ機能に制約を受けます。C言語で文字データを扱う場合、そのデータの終わりに0を入れるという規則になっていますが、ある種のコンピュータはこの処理方法では実行速度が極端に遅くなってしまいます。アセンブラ言語は、このようなコンパイラ言語の制約に振り回されず、そのコンピュータ独特の**チューニング**(*1)を行うことができるのです。

実際、ある調査によると組込み用プログラム（家電製品や自動車のエンジンなどを制御するプログラム）の80%はC言語ですが、そのうち、半数がアセンブラ言語を併用しています。また、10%はアセンブラ言語のみによる開発です。

### (2) 高水準言語を使う時に、その出力される機械語のパターンが連想できるようになる

高水準言語でプログラムを書く時に、そのプログラムが機械語に変換されるパターンが分かると、割と効率の良いプログラムが書けるようになります。つまり、高水準言語でプログラムを書く時の、知識の幅が広がるのです。

それでは、次の疑問として、COMET IIのような現実には存在しないコンピュータではなく、現実のコンピュータの機械語あるいはアセンブラ言語を学習すれば良いではないかと思う人もいるかも知れません。

COMET IIは、前述のように情報処理技術者試験のために作られた仮想的なコンピュータです。それも、現実のコンピュータと比べると極端に単純化されたコンピュータです。現在の代表的なコンピュータであるマイクロプロセッサとCOMET IIの比較を表1-1に示します(*2)。これを見ると、いかにCOMET IIが単純化されたコンピュータであるかが理解できるでしょう。しかし、現実の複雑な機構を備えたコンピュータを初心者が学習することは、とても大変です。そこで、初心者学習用としてコンピュータのエッセンスを抜き出して作成されたのがCOMET IIなのです。

それならば、こんな単純なものを理解しても何の勉強にもならないではないかと思う人もいるかも知れません。しかし、COMET IIは、実際のコンピュータの本質的な部分だけを抜き出しているので、

(*1) レーシングカーのエンジンをいろいろ設定を変えて、ある条件で最高の出力ができるように調整することをチューニングと言います。これと意味はまったく同じで、プログラムの実行順序やデータの配置をいろいろと変えて、実行するコンピュータのハードウェア特性に合わせて最高の性能を出せるように改造することをチューニングと言います。

(*2) 表1-1の内容を見ても、そこに何が書いてあるかまったくわからない人も多いでしょう。ここで、大切なことは、COMET IIというのは、実在のコンピュータをその基本的なところだけを抜き出して作成されたコンピュータであるということを理解してもらうことだけなので、細かなことは分からなくても気にしないで下さい。

基本的なコンピュータの理解は、COMET IIを学習するだけでも十分にできるのです。逆に現実的でやっかいな問題に振り回されずに、効果的な学習ができるという利点があります。

また、COMET IIが仮想コンピュータであるもう一つの理由として、このコンピュータが情報処理技術者試験用であることが挙げられます。試験の公平性を保つためには、特定メーカのコンピュータを取りあげる訳にはいきません。つまり、一部の人が有利にならないようにしているのです。

**表 1-1　代表的なマイクロプロセッサとCOMET IIとの比較**

| | | 代表的なマイクロプロセッサ | COMET II |
|---|---|---|---|
| 扱えるデータ | | 8,16,32ビット | 16ビット |
| 汎用レジスタの数 | | 32個(RISCの場合) | 8個 |
| 仮想記憶 | | 有 | 無 |
| メモリ保護 | | 有 | 無 |
| アドレス空間 | | 64T(テラ)バイト($2^{46}$,80386の例) | 65535語($2^{16}$語、1語2バイト) |
| 命令セット | データ転送命令 | ○ | ○ |
| | 整数演算 | ○ | △(乗算、除算なし) |
| | 論理演算 | ○ | ○ |
| | 浮動小数点演算 | ○ | × |
| | 10進演算 | ○ | × |
| | 制御 | ○ | ○ |
| | サブルーチン | ○ | ○ |
| | I/O | ○ | △(基本機能のみ) |

**♥CASLDVのインストールとアンインストール♥**

CASLDVのインストール方法は、本書に付属のCD-ROMのCASLDVマニュアルにも詳しく記載されていますが、ここでも簡単に説明することにします。

CASLDVをインストールするには付属のCD-ROMのInstallフォルダの中のSetup.exeを実行してください。すると、インストーラが起動します。後は、インストーラの指示に従ってインストールしてください。その中で製品ライセンス契約の部分がありますのでその内容に同意される方だけご使用ください。また、インストールするフォルダを聞いてくるので指定してください。指定したフォルダの中に、tempとbinというフォルダが作成され、CASLDVの本体CASLDV.exeがbinの中にコピーされます。このショートカットをディスクトップ上に作ると便利です。

CASLDVをアンインストールするには、「スタート」メニューから「設定」を選び、「コントロールパネル」を表示し、「アプリケーションの追加と削除」を実行し、CASLDVを選んでください。なお、場合によって作成したフォルダが全て削除できないこともあります。そのときは手作業で削除してください。

## 1.3 コンピュータのハードウェア構成

一般にコンピュータのハードウェアは、入力装置、出力装置、演算装置、制御装置、主記憶装置から構成されています。これらの装置をまとめてコンピュータの**五大装置**と呼びます。この構成は、初期のコンピュータから現在のパソコンに至るまで変わっていません。

それでは、現在、最も多く使われているパソコンに対応して直感的に説明しましょう。**入力装置**はデータを入力するための装置で、キーボードやマウスが対応します。入力されたデータは主記憶装置に転送され、そこで記憶されます。**出力装置**は、主記憶装置内のデータを出力するための装置で、ディスプレイやプリンタが対応します。演算装置と制御装置は、**CPU**(Central Processing Unit)と呼ばれる超LSIの中に格納されています。現在の多くのパソコンには、**Pentium**(ペンティアム)と呼ばれるインテル社製のCPUあるいはその互換CPUが搭載されています。**演算装置**は、文字通り演算を行う回路です。COMET IIでは、整数の加算・減算・ビット操作の演算が用意されていますが、ほとんどのコンピュータでは、整数および実数の加算・減算・乗算・除算が用意されています。**制御装置**は、機械語の指令に従い、各装置に制御信号を送るための回路です。**主記憶装置**は、プログラムやデータを記憶するための装置で、パソコンの筐体（きょうたい：本体）の中に格納されたLSIによって構成されます。なお、主記憶装置は、単に記憶装置と呼ぶこともあります。

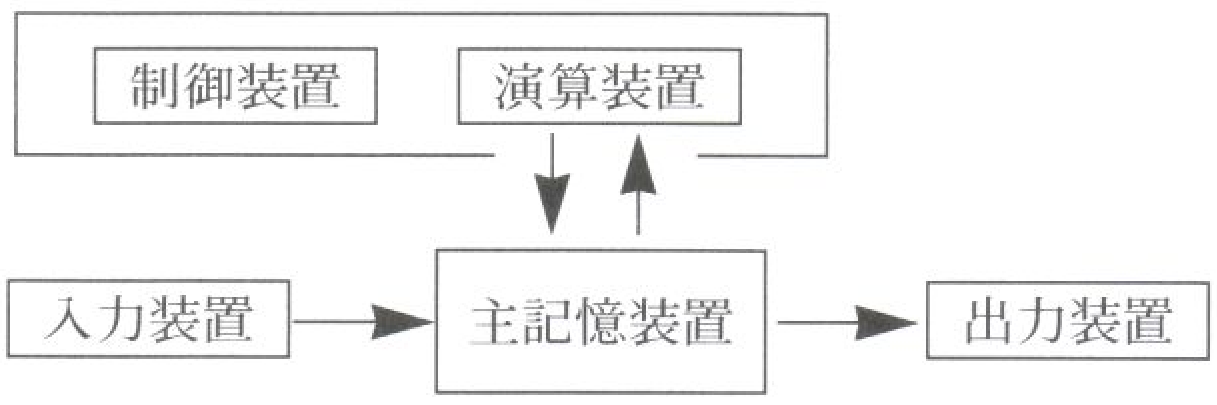



**図1-6 コンピュータのハードウェア(五大装置)**

ただし、これだけではコンピュータは作動しません。まず、プログラムやデータを永続的に保存するための**補助記憶装置**が必要になります。最も代表的なものは、ハードディスクでしょう。この他に、CD-ROMドライブやフロッピードライブなどがあります。補助記憶装置は、データの入出力という観点から見れば、入力装置および出力装置と考えることもできます。

プログラムを動作させるには、入力装置を介して、プログラムと計算に必要なデータをあらかじめ主記憶装置に入れておき、コンピュータはプログラムをCPUの中の制御装置に一命令ずつ取り込み、その機械語を解読して実行します。一つの命令の実行が終わると、次の命令を取り込んで解読して実行します(*1)。

さて、コンピュータは、主記憶装置に格納されたプログラムを逐一実行していくと説明しました。しかし、プログラムは初めから主記憶装置に格納されているわけではありません。現在のコンピュータは、主記憶装置に半導体を用いていますので、電源が入っていないと情報を貯えることができません。つまり、電源投入時には主記憶装置には何のデータもプログラムも格納されていません。

実際に、パソコンの電源を投入しても、直ちにコンピュータが利用可能になるわけではありません。パソコンの電源を入れると、まず、コンピュータ内部に組込まれた非常に小さなプログラムが主記憶

(*1) 現在のコンピュータは、実行速度を上げるために、**パイプライン処理**という方法を採用しています。この方法では、機械語の命令は、(1)命令の取り込み、(2)解読、(3)実行というステップを同時に行うことにより処理効率を高めます。たとえば、プログラムの中に「命令1、命令2、命令3」という3つの機械語があると、命令1の実行中に、命令2の解読が行われており、さらに同時に命令3の取り込みが行われています。

装置に取り込まれます。そして、そのプログラムが起動され、**オペレーティングシステム**(Operating System, 略してOSとも呼びます)という巨大な管理プログラムを主記憶装置に取り込むのです。この過程を**ブートストラップ**と呼びます(*1)。

さて、このオペレーティングシステムは、パソコン用としてはマイクロソフトのWindows 98, Windows 2000、Unix系のLinux, Free BSDなどいくつか種類があります。通常、プログラムは、このオペレーティングシステムの管理下で動作します。たとえば、Windows 98上で、WORD(ワードプロセッサ), EXCEL(表計算ソフト), ソリティア(ゲーム)を実行させていると、これらのプログラムが、オペレーティングシステムとともに主記憶装置上に置かれるのです(*2)。

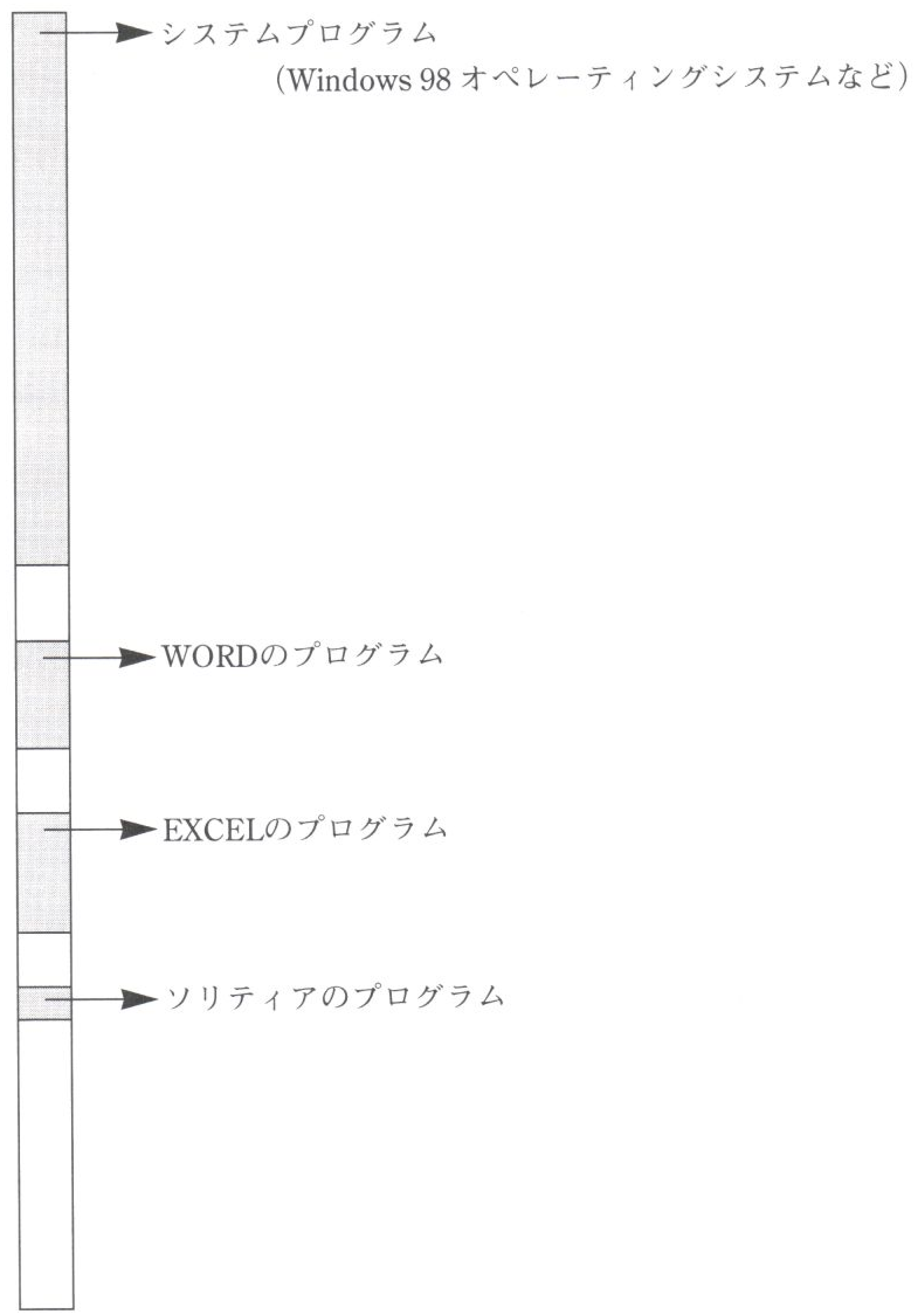



**図1-7 実行中のプログラムが主記憶装置に格納されている様子**

(*1) JISでは、ブートストラップとは、「命令の集合であって、完全な計算機プログラムが記憶装置に入り終わるまで、後続の命令をロードするもの」と定義されています。

(*2) これらのプログラムは、実際のパソコンでは、ハードディスク装置の一部を記憶装置に見立てた仮想記憶装置の中に格納されます。

## 1.4 本書で提供するソフトウェアについて

さて、世の中には、COMET IIコンピュータは実在しないと前述しました。それならば、なぜ、CASL IIアセンブラ言語で書かれたプログラムをCASL IIアセンブラでCOMET IIの機械語に変換して実行できるのでしょうか。それは、COMET IIの機械語を実行するコンピュータと同じ動作を行うプログラムが用意されているからです。これを**COMET IIシミュレータ**と呼びます。

本書では、次のソフトウェアを読者のために用意しました。ただし、本書に付属のCD-ROMの中に格納されているソフトウェアはWindows系（Windows 95/98/NT/2000）のCASLDVだけです。CASLDVは、その中にCASL IIアセンブラとCOMET IIシミュレータを含んでいます。

CASLDV以外のバージョンを入手希望の方は本書の序文をご覧下さい。

**表1.2 本書で提供するソフトウェアと動作環境**

| 動作可能 OS | Windows 95 | Windows 98 | Windows NT 4.0 | Windows 2000 | Linux | Free BSD | Solaris |
|---|---|---|---|---|---|---|---|
| CASL II アセンブラ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| COMET II シミュレータ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| CASL II 開発環境<br>CASLDV(含デバッガ) | ○ | ○ | ○ | ○ | × | × | × |

CASL IIアセンブラは、CASL IIプログラムをCOMET IIの機械語に変換するソフトウェアです。これは、コマンドを打ち込む形式で利用します。COMET II シミュレータは、CASL IIアセンブラによって生成された機械語ファイルを実行するためのソフトウェアです。CASLDVと名付けられたCASL II開発環境は、CASL IIのプログラムを作成し、実行するまでのすべての段階を一つのウィンドウの中で行うことができるものです。なお、これにはエディタやデバッガも付いています。

CASLDVの簡易マニュアルは本書の巻頭に記載されていますが、CD-ROM中に本格的なマニュアルがありますので、詳しくはそちらをご覧下さい。Windowsをお使いの方は、CASL II開発環境ですべてのことができるので、本書に付属のCASLDVをお使いいただくとよろしいかと思います。

なお、CASLDVをパソコン教室などで使用するために、システム環境設定をできないようにしたバージョンもあります。入手希望の方は本書の序文をご覧下さい。

# 総合問題1

**[演習1-1]** 次の文章を読み、正しい文章には○、誤りのある文章には×を付けてください。
誤りのある場合には、その理由も指摘してください。また、条件によってどちらともいえない場合には△を付けて、その条件を説明してください。

(1) コンピュータが実際に解釈して実行することのできる言語は機械語だけである。

(2) 機械語にはJIS規格があって、どのコンピュータでも同じ機械語が通用する。

(3) アセンブラ言語で書かれたプログラムは効率が良いのであるから、すべてのプログラムはアセンブラ言語で開発すべきである。

(4) 一般にアセンブラ言語でプログラムを書くよりも高水準言語でプログラムを書いた方が人間にとっての労力がはるかに少ない。

(5) 基本的に、アセンブラ言語は、そのコンピュータの機械語に1対1に対応している。

(6) アセンブラ言語を使えば、どんなプログラムであってもそのコンピュータのすべての種類の機械語命令を実行できる(やや難)。

(7) プログラムを実行させるには、必ず主記憶装置に配置しなければならない。

(8) 以前は、プログラム内蔵方式が主流であったが、今はそうではない。

(9) アセンブラ言語で書かれたプログラムは、アセンブルという処理を経るので、その実行効率はほんのわずかではあるが機械語に劣る。

(10) C言語のコンパイラの多くは優れた最適化機能を有しているので、アセンブラ言語で作成されたプログラムよりも常に効率が良くなる。

(11) 同じWindows 98を搭載したパソコンでも、メーカが異なれば異なるコンピュータなので同じ機械語は動作しない。

(12) 同じCPUを搭載したパソコンであれば、Windows系OSでもUnix系OSでも、同じ機械語プログラムが動作する(やや難)。

*Memorandum*

# 第2章 数の表現とその演算

## 2.1 アセンブラ言語と数の表現

アセンブラ言語を理解するためには、ハードウェアの理解が必要です。ハードウェアの内部を覗くには、2進法や8進法、16進法といった数の表現方法を理解することが重要になります。日頃、10進法に馴染んでいる私たちにとって、これらの表現方法は、慣れないと面倒です。しかし、逆に慣れてしまうと、アセンブラ言語で書かれたプログラムを読む際に、そこで書かれた数値が16進法で書かれた数値でないとなぜかしっくりいかなくなります。たとえば、256という10進法で表現された数値よりも、16進法で00FFと表現された数値の方が読みやすく感じるようになります。とはいっても、みなさんは、日常とは違う数値で書かれた世界と初めて接する訳ですから、これが、アセンブラ言語を習得する最初の難関であると思って頑張ってください。

なお、2進法で記述されている数値を2進数、8進法で書かれている数値を8進数、10進法で書かれている数値を10進数、16進法で書かれている数値を16進数と呼ぶことにします。

また、数の表現を学習する際に大切なことは、「ただ単に暗記することではなく、原理を理解する」ことです。そうすれば、数の表現に関する知識は自然と身についていきます。

**♥CASLDVの起動♥**

まず、CASL IIのプログラムを格納するフォルダを作成してください。CASLDVを起動するには、「スタート」メニューから「プログラム」を選択し、その中から「CASLシミュレータ」→「CASLシミュレータ」を選んでください。ショートカットがある場合には、それをダブルクリックするかそこにファイルをドラッグすると起動します。

起動すると、左上・右上・下の部分に3つのペイン(画面)が表示されます。これをアセンブラリストペイン(左上)、エディタペイン(右上)、インフォメーションペイン(下)と呼びます。エディタペインにCASL IIのプログラムを入力してください。ファイル名は「～.casl」として、ディスク装置などに保存してください。

「アセンブル」メニューから、「アセンブル(CASL完全準拠)」を選び、アセンブラを起動してください。エラーがある場合、インフォメーションペインにエラーメッセージが表示されるので、それをダブルクリックしてください。エディタの該当する行が反転し、位置を知らせてくれます。エラーがなくなると、アセンブラリストペインにアセンブラリストが表示されます。なお、アセンブラの起動は、ツールバー上のボタンでも行えます。

プログラムを実行させるには、「実行」メニューから「実行開始」を選んでください。インフォメーションペインにトレース結果が表示されます。なお、「実行開始」もツールバー上に対応するボタンがあります。

## 2.2 10進法

一般に、私たちは**10進法**を用いています。その理由は、私たちの手の指の合計が10本であったからだと言われています。10進法では、0, 1, 2, と数えていきます。そして、9の次が1つ繰り上がって10になります。そして、さらに、10, 11, 12, と数えていき、19の次が20となります。ちょうど、10進数の各桁には0〜9の10個の数字のどれかが対応することになります。

さて、10進法では、12345という数値は、

$$1\times10000+2\times1000+3\times100+4\times10+5$$

というように表現できます。つまり、12345という数値は、10000が1個、1000が2個、100が3個、10が4個、1が5個ある数ということになります。

これをもう少し、数学的に書くと

$$1\times10^{4}+2\times10^{3}+3\times10^{2}+4\times10^{1}+5\times10^{0}$$

と書くことができます。各桁の$10^k$をその桁の重みと呼びます。

12345という10進法による数値を図的に表現すると次のようになります。

| 10進数値の桁<br>桁の重み | | 1<br>10000 (=$10^4$) | | 2<br>1000 (=$10^3$) | | 3<br>100 (=$10^2$) | | 4<br>10 (=$10^1$) | | 5<br>1 (=$10^0$) |
|---|---|---|---|---|---|---|---|---|---|---|
| その値 | | 10000 | | 2000 | | 300 | | 40 | | 5 |
| 合計値 | 12345 = | 10000 | + | 2000 | + | 300 | + | 40 | + | 5 |

前述のように10進法の場合、各桁の重みは$10^k$となります。一般に、n進法では、k桁目の重みは$n^k$となります。

### ポイント

10進法では各桁の重みは$10^k$となります。

$$12345 = 1\times10^4+2\times10^3+3\times10^2+4\times10^1+5\times10^0$$

### 演習問題2-2

私たちは、日常的に10進法を用いていますが、例外的に10進法以外の進法も身近に用いています。どのような進法を何に用いているか例を挙げて説明してください。

## 2.3 2進法

### 2.3.1 2進法の表現

それでは、まず、**2進法**について考えてみましょう。2進法では、0と1の2つしか数字を使いません。ゼロは2進法でも0と表現しますし、1も同様に2進法で1と表現します。しかし、10進数と2進数の表現が同じなのは、この2つ(0と1)だけです。それでは、10進法の2は、2進法ではどのように表現するのでしょうか。数が0と1しかないので、1桁では2という値は表現できません。桁が1つ繰り上がって、10という数で2を表現します。

表2-1に、10進数の0～8に対応する2進法の数値を示します。数字が少ないために、10進数の8を表現するのに2進数では4桁も必要になってしまいます。各桁の繰り上がりの様子を観察し、何も見ないでもこれらの数値を書けるようにしてください。

ここで大切なことは、2進法の各数値を暗記するのではなく、桁の繰り上がりの規則(0の次が1となり、その次に繰り上がる)を理解し、それを再現できるようにすることです。

表 2-1　0～8の2進法の表現

| 10進数 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 2進数 | 0 | 1 | 10 | 11 | 100 | 101 | 110 | 111 | 1000 |

さて、それでは、この2進法で表現された数値、いわゆる2進数について考えてみましょう。

10101という2進数は、次のように解釈されます。

$$1\times2^4+0\times2^3+1\times2^2+0\times2^1+1\times2^0=1\times16+0\times8+1\times4+0\times2+1\times1=16+4+1=21$$

すなわち、k桁目の重みが、$2^k$と解釈されます。

| | | | | | | |
|---|---|---|---|---|---|---|
| 2進数値の桁 | | 1 | 0 | 1 | 0 | 1 |
| 桁の重み | | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| | | 16 | 8 | 4 | 2 | 1 |
| その値 | | 16 | 0 | 4 | 0 | 1 |
| 合計値 | 21 = | 16 + | 0 + | 4 + | 0 + | 1 |

2進数で10101という数は、私たちの分かりやすい10進数では、16+4+1=21という数になります。なお、数値が2進数であることを明示的に示すために、数値の最後に(2)を付けて、10101(2)と表現することもあります。一般に、n進数の場合、(n)を付けますが、10進数の場合には、通常省略します。

表 2-2　n進数の示し方

| 進　数 | 例 |
|---|---|
| 2進数 | 101101 (2) |
| 8進数 | 55 (8) |
| 10進数 | 45 |
| 16進数 | 2D (16) |

表2-3に0から63までの2進数とそれに対応する10進数の表を示します。各桁の位置関係が分かるように表2-3では全ての2進数を7桁で示し、先行するゼロを明示します。

**表 2-3　0から63までの2進数とそれに対応する10進数**

| 10進 | 2進 | 10進 | 2進 | 10進 | 2進 | 10進 | 2進 |
|---|---|---|---|---|---|---|---|
| 0 | 000000 | 16 | 010000 | 32 | 100000 | 48 | 110000 |
| 1 | 000001 | 17 | 010001 | 33 | 100001 | 49 | 110001 |
| 2 | 000010 | 18 | 010010 | 34 | 100010 | 50 | 110010 |
| 3 | 000011 | 19 | 010011 | 35 | 100011 | 51 | 110011 |
| 4 | 000100 | 20 | 010100 | 36 | 100100 | 52 | 110100 |
| 5 | 000101 | 21 | 010101 | 37 | 100101 | 53 | 110101 |
| 6 | 000110 | 22 | 010110 | 38 | 100110 | 54 | 110110 |
| 7 | 000111 | 23 | 010111 | 39 | 100111 | 55 | 110111 |
| 8 | 001000 | 24 | 011000 | 40 | 101000 | 56 | 111000 |
| 9 | 001001 | 25 | 011001 | 41 | 101001 | 57 | 111001 |
| 10 | 001010 | 26 | 011010 | 42 | 101010 | 58 | 111010 |
| 11 | 001011 | 27 | 011011 | 43 | 101011 | 59 | 111011 |
| 12 | 001100 | 28 | 011100 | 44 | 101100 | 60 | 111100 |
| 13 | 001101 | 29 | 011101 | 45 | 101101 | 61 | 111101 |
| 14 | 001110 | 30 | 011110 | 46 | 101110 | 62 | 111110 |
| 15 | 001111 | 31 | 011111 | 47 | 101111 | 63 | 111111 |

この表の数値ををすべて覚える必要はありません。2進数を10進数に変換する必要がある場合には、まず、ノートに次に示す2進数の各桁の重みを右から1,2,4,8,…と書きます。

... 2048　1024　512　256　128　64　32　16　8　4　2　1

これらは、1から始めて数を2倍ずつしていったものです。もちろん、これらの数値はいちいち計算して書くのではなく、即座に書けるようにしておく必要があります。しかし、これもいちいち暗記するのではなく、何度か1を2倍していく計算を行えば自然に覚えられます。

さて、たとえば、10011011010(2)という数を10進数に変換することを考えましょう。まず、この各桁の数値(0か1)を重みの下に書きます。

| ... 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |

数字が1の部分だけを下に転記し、その合計を加えます。

| ... 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | |
| | 1024 | ＋ | | 128 ＋ | 64 | ＋ | 16 ＋ | 8 | ＋ | 2 | | = 1242 |

この合計値1242が求める10進数となります。この計算方法だけを覚えておけば、暗記に頼る必要はありません。

ただし、２進数の表現に関して、いくつかの特徴を理解しておく必要があります。表2-3をみると、奇数のときに最下位桁が1になり、偶数のときに最下位桁が0になっていることに気付きませんか。これは、2進数（$b_5 b_4 b_3 b_2 b_1 b_{0\,(2)}$）の定義

$$
\begin{aligned}
& b_5 \times 2^5 + b_4 \times 2^4 + b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 \\
= & b_5 \times 32 + b_4 \times 16 + b_3 \times 8 + b_2 \times 4 + b_1 \times 2 + b_0 \times 1
\end{aligned}
$$

を思い出せばすぐに理解できるでしょう。2桁め($b_1$)以降の重みは必ず2の倍数になっているので、その数が偶数か奇数かは、最下位桁$b_0$の値で決まるのです。$b_0$が1ならその数は奇数であり、$b_0$が0ならその数は偶数です。

同様に、下2桁が00になっているとその数は4の倍数であり、下3桁が000になっていると8の倍数であることが分かるでしょう。一般に、2進数の下n桁すべてが0になっていると、その数は$2^n$の倍数となります。このことを知っておくと、奇遇判断(数が偶数か奇数の判断)や$2^n$の倍数かどうかの判断を、後述するビット操作を使って簡単に行うことができます。

## ポイント

2進数10011011010 (2) を10進数に変換するには「2の重み($2^k$)」を書く

| ...2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | | |
| | 1024 | + | | 128 | + 64 | + | 16 | + 8 | + | 2 | | = | 1242 |

**♥CD-ROMの内容♥**

本書に付属のCD-ROMには、ソフトウェア(CASLDV)とそのマニュアルだけでなく、本書で提供しているすべてのソースプログラムと演習問題で出題されたすべてのソースプログラムが含まれています。Sourceフォルダの中には各章に対応したフォルダがあり、その中に本書で述べられているプログラムのソースファイルがあります。Answerフォルダの中にも各章に対応したフォルダがあり、その中に演習問題のプログラムがあります。「高度なサンプル」というフォルダには、本書では言及していないが読者のためになる高度なレベルのプログラムが入っています。そこには、HANOI.casl(ハノイの塔)、PRIME.casl(素数)、SORT.casl(ソート、並び替え)のプログラムがありますので、参考にしてください。

## 演習問題2-3-1

(1) 次の2進数を10進に変換してください。

(a) 1010

(b) 111011

(c) 10101011

(2) 次の2進数値が「奇数」、「2の倍数(=偶数)」、「4の倍数」、「8の倍数」のどれに相当するか次の表を○×(○：相当する、×：相当しない)で埋めてください。

| 数値 | 「奇数」、「2の倍数(=偶数)」、「4の倍数」、「8の倍数」 |
|---|---|
| 11011 | |
| 100000 | |
| 111110 | |
| 110100 | |
| 100011 | |

### 2.3.2　2進法の計算方法

それでは2進法の計算方法について考えてみましょう。10進数では1＋1=2となりますが、2進数では1＋1=10(正確に書くなら、$1_{(2)}+1_{(2)}=10_{(2)}$)となります。それでは、もっと桁の多い2進数の加算はどうするのでしょう。

まず、3＋4を考えてみましょう。この答えは7ですが、2進数では次のように計算します。2進数では、$11_{(2)}$が3、$100_{(2)}$が4であることは、もう理解していますよね。桁を合わせるために、次の式では$11_{(2)}$を$011_{(2)}$としていることに注意してください。

$$\begin{array}{r} 0\ 1\ 1 \\ +\ 1\ 0\ 0 \\ \hline 1\ 1\ 1 \end{array}$$

上記のように、2進数の各桁を加えます。この場合、すべての桁が1となり、$111_{(2)}$となります。この数は10進数では7となり、計算が正しいことがわかります。

それでは、20＋40=60の計算をしてみましょう。$20=10100_{(2)}$, $40=101000_{(2)}$ですから、次のようになります。

$$\begin{array}{r} 0\ 1\ 0\ 1\ 0\ 0 \\ +\ 1\ 0\ 1\ 0\ 0\ 0 \\ \hline 1\ 1\ 1\ 1\ 0\ 0 \end{array}$$

これも同様に各桁を加えればよく、その答えは$111100_{(2)}=60$となります。

それでは、13＋9=22を計算してみましょう。$13=1101_{(2)}$、$9=1001_{(2)}$です。1＋1の計算を行う桁が1桁めと4桁めにあります。この場合、繰り上がって$10_{(2)}$となりますので、次の桁にその部分を足します。

```
      1   1   0   1
 ＋   1   0   0   1
 ─────────────────────
     10   1   0  10    各桁の合計
  1↙         1↙        繰り上がり部分
 ─────────────────────
  1   0   1   1   0
```

結果は、10110(2)=22となります。

次に、15＋1=16はどうなるでしょうか。15=1111(2), 1=0001(2)ですから、次のようになります。

```
      1   1   1   1
 ＋   0   0   0   1
 ─────────────────────
      1   1   1  10
             1↙        繰り上がり部分
 ─────────────────────
      1   1  10   0
         1↙            繰り上がり部分
 ─────────────────────
      1  10   0   0
      1↙               繰り上がり部分
 ─────────────────────
     10   0   0   0
  1↙                   繰り上がり部分
 ─────────────────────
  1   0   0   0   0
```

答えは、10000(2)=16となります。このように、繰り上がりは次々に伝播していくので注意が必要です。

なお、一般的な2進数の演算では、表現できる桁数に制限はありませんが、COMET IIの場合、16桁までの2進数しかデータを保持できない(2.3.3項参照)ので、計算結果がその桁数を超える値になった場合にはその部分は無視されます。計算結果の桁数が表現できる桁を越えることを**オーバーフロー**と呼びます。

```
       ←──16桁──→
       1111111111111111
 ＋                   1
 ───────────────────────
     1 0000000000000000
     ↑            │
  オーバー         │
  フローした       │
  部分は無視       │
  される           ↓
       0000000000000000 ← 最終的な答え
       ←──16桁──→
```

なお、減算については、2.3.6項「減算の仕組み」で説明します。

## 演習問題2-3-2

2進法で次の計算を行ってください。

(1) $11_{(2)}+10101_{(2)}$

(2) $1001_{(2)}+110111_{(2)}$

### 2.3.3 2進数とハードウェアの関係

それでは、なぜ、2進数などという面倒な表現方法を学習するのでしょうか。それは、コンピュータのハードウェアと密接な関係があります。現在のコンピュータは、電流のON/OFFという2つの状態でデータを表現します。

このON/OFFが、ちょうど2進数の0と1に対応しているのです。COMET IIのコンピュータの内部では、前述の21という数値は、$0000000000010101_{(2)}$という形で記憶されます。

通常は、0がOFF, 1がONに対応します。

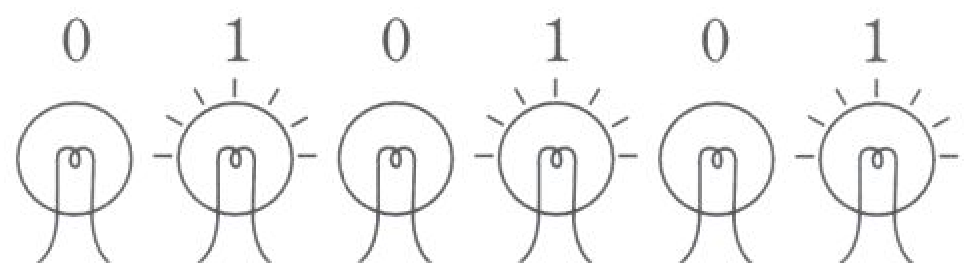



図2-1 電流の流れとビットのON/OFF

2進数の1桁がちょうどこのON/OFFに相当しますが、この情報を保持した桁のことを**ビット**(bit)と呼びます。ON/OFFに相当する1ビットでは、0と1の2つのデータを表現できます。2ビットでは、00, 01, 10, 11の4つのデータを表現できます。3ビットでは、000, 001, 010, 011, 100, 101, 110, 111の8つのデータを表現できます。一般に、nビットでは、$2^n$個のデータを表現できます。

8ビットのまとまりを**1バイト**(byte)と呼びます。1バイトは$2^8$=256通りのデータを表現できます。COMET IIは、2バイトすなわち16ビットまでのデータを保持できますので、$2^{16}$=65536通りのデータを表現することができます。

なお、前述のように、COMET IIでは、2進数値を16ビットで表現しますが、その各ビットには次のような**ビット番号**がついています（符号については、2.6節で説明します）。

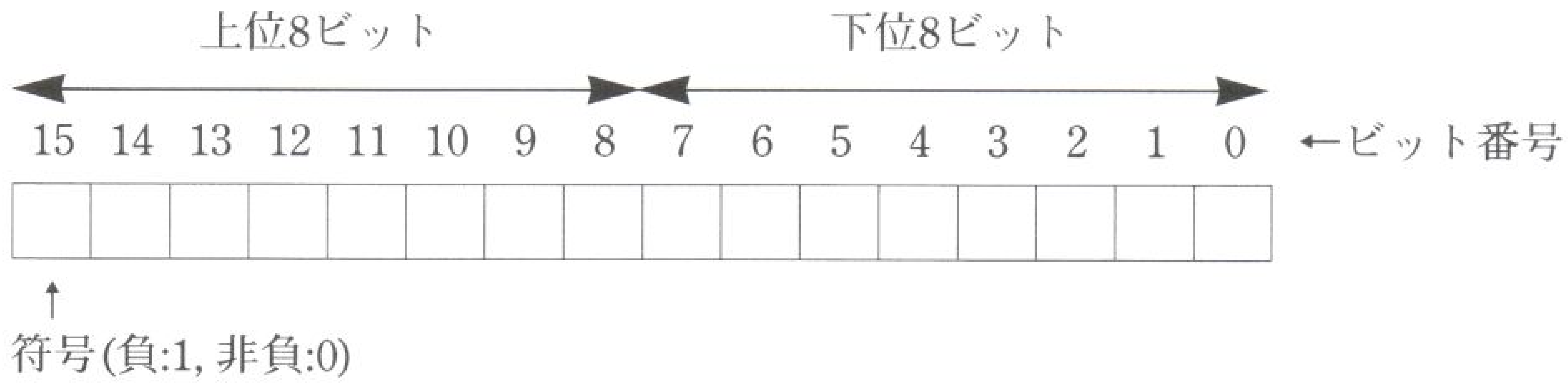



図2-2 COMET IIにおけるビット構成

ビット番号0のビットを**最下位ビット**、ビット番号15のビットを**最上位ビット**と呼ぶことがあります。

## 2.4 8進法

2進数値は、コンピュータ内の電源のON, OFFにダイレクトに対応するので、コンピュータの内部のデータの様子を、そのまま表現できることがお分かりいただけたかと思います。しかし、2進数では、使用する数字が0と1だけの2つしかないので、どうしても数の桁数が多くなります。そこで実際には、8進数、16進数を使います。CASL IIでは8進数を表現することができませんが、本書では8進数も説明することにします。

**8進数**では、0から7までの数値を使ってデータを表現します。8進数では、$0_{(8)}$, $1_{(8)}$, $2_{(8)}$, $3_{(8)}$, $4_{(8)}$, $5_{(8)}$, $6_{(8)}$, $7_{(8)}$と数えていき、$7_{(8)}$の次に繰り上がりが起きて、$10_{(8)}$となります。そして、$11_{(8)}$, $12_{(8)}$, ..., $17_{(8)}$と数えていき、$17_{(8)}$の次に$20_{(8)}$となります。

それでは、$123_{(8)}$という数値は、10進法ではどのような値になるのでしょう。$123_{(8)}$は、$1\times8^2+2\times8^1+3\times8^0=1\times64+2\times8+3\times1=64+16+3=83$となり、10進数では83ということになります。表2-7に、10進数、8進数、16進数の0から63までの一覧表を示すので参考にしてください。

8進数と2進数の間には、とても単純な関係があります。8進数の各桁は、ちょうどそれに対応する2進数を下位から順に3桁ずつ区切った値に対応します（3桁に満たない場合は0を補います）。たとえば、$1110011001010_{(2)}$は$16312_{(8)}$となります。

```
┌─ 3桁に満たないので0を補う
▼
001  110  011  001  010
 1    6    3    1    2
```

2進数値を3桁ごとに区切ると、その中には$2^3$=8個のデータがあり、これがちょうど8進数の0〜7の数値に対応しています。そのため、2進数を3桁ごとに区切ると、ちょうど、8進の各桁の重みに一致するので、このような単純な関係になるのです。

この関係から、8進数の0から7までのビットパターンを覚えておけば、前述の$16312_{(8)}$という8進数がどのようなビットパターンを形成するかはすぐに思い浮かべられると思います。

**表 2-4 8進数のビットパターン**

| 桁 | ビットパターン |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

## 演習問題2-4

次の数値を10進数に変換してください。

(1) $13_{(8)}$
(2) $57_{(8)}$
(3) $1234_{(8)}$

**♥CASLDVにおけるプログラムの実行♥**

CASLDVでは、次の方法でプログラムの実行ができます。

(1) 高速実行
(2) トレース実行
(3) 1命令ずつのステップ実行
(4) ブレークポイントまでの実行

(1)と(2)はトレースを行うか行わないかの違いだけで実質的な違いはありませんが、実行速度がかなり異なります。トレースの指定は、「オプション」メニューから「実行時オプションの指定」を選び、そのダイアログの「さまざまな実行時オプション」グループの中の「実行時とレースを出力する」で行います。これがチェックされていれば実行時とレースを行い(規定値)、チェックされていなければ行いません。いずれにせよ、(1)、(2)は「実行」メニューの「実行開始」によって行います。なお、このトレース機能は、次で説明する(3)、(4)でも有効です。

(3)と(4)は実行の様子をグラフィカルに示すモードです。このモードでは、まず、「実行」メニューの「ステップ実行開始」を選択します。この後で、↓(矢印)キーを押すと1命令分ステップ実行します。ステップモードの時に、「実行」メニューから「ブレークポイントまで実行」を選ぶと設定されたブレークポイントまで一気に実行します。ブレークポイントを設定するには、アセンブラリストペインに示されたアセンブラリストの機械語のアドレスをマウスでクリックします。解除するには、そこを再びクリックします。なお、ブレークポイントはいくつでも設定できます。

(3)と(4)のモードでは、プログラムの実行の様子がグラフィカルに示されるため、実行のメカニズムをより深く理解できるようになると思います。

なお、実行に関するメニューは、ツールバー中のボタンにもありますので、実際にはそちらをお使いください。

## 2.5 16進法

さて、**16進法**の場合も同様に考えます。16進数では、16個の数字を使います。そのため、0から9までの10個の数字とA, B, C, D, E, Fまでの6個の数字の合計16の数字を使います(*1)。

表2-5に、16進数の0〜9、A〜Fの各数値に対応する2進数、10進数を示します。

16進数では、$0_{(16)}$, $1_{(16)}$, $2_{(16)}$, $3_{(16)}$, $4_{(16)}$, $5_{(16)}$, $6_{(16)}$, $7_{(16)}$, $8_{(16)}$, $9_{(16)}$と数えていき、この次に、$A_{(16)}$となります。これは繰り上がりが起きたのではありません。9の次に対応する数字として(10進数では10ですが)、Aという文字を数字として用いるのです。そして、Aの次の数は$B_{(16)}$となります。Bは、10進数では11に相当します。そして、$C_{(16)}$, $D_{(16)}$, $E_{(16)}$, $F_{(16)}$まで数えていき、この次に繰り上がりが発生し、$10_{(16)}$になります。そして、$11_{(16)}$, $12_{(16)}$, ..., $1E_{(16)}$と数えていき、 $1F_{(16)}$の次が$20_{(16)}$になります。

それでは、$123_{(16)}$という数値は、10進法ではどのような値になるのでしょう。$123_{(16)}$は、$1\times16^2+2\times16^1+3\times16^0=1\times256+2\times16+3\times1=256+32+3=291$となり、10進数では291ということになります。表2-7に、10進数、8進数、16進数の0から63までの一覧表を示すので参考にしてください。

16進数と2進数の各桁の間には、とても単純な関係があります。16進数の各桁は、2進数を4桁ずつ区切った数になります（4桁に満たない場合は0を補います）。

```
┌── 4桁に満たないので0を補う
↓
0101 1100 1100 1010
 5    C    C    A
```

上記のように、$101110011001010_{(2)}$は、$5CCA_{(16)}$となります。

**表 2-5　16進数字の表現**

| 16進数 | 10進数 | 2進数 | 16進数 | 10進数 | 2進数 |
|---|---|---|---|---|---|
| 0 | 0 | 0000 | 8 | 8 | 1000 |
| 1 | 1 | 0001 | 9 | 9 | 1001 |
| 2 | 2 | 0010 | A | 10 | 1010 |
| 3 | 3 | 0011 | B | 11 | 1011 |
| 4 | 4 | 0100 | C | 12 | 1100 |
| 5 | 5 | 0101 | D | 13 | 1101 |
| 6 | 6 | 0110 | E | 14 | 1110 |
| 7 | 7 | 0111 | F | 15 | 1111 |

16進数は機械語の世界では特によく使います。16進の数値と2進の数値の対応づけは、完全に頭の中に入れておく必要があります。たとえば、$F_{(16)}$は$1111_{(2)}$を意味し、$A_{(16)}$は$1010_{(2)}$を、$8_{(16)}$は$1000_{(2)}$を意味します。

このことを頭の中に入れておけば、16ビットの数値の場合、$FFFF_{(16)}$はすべてのビットが1、$AAAA_{(16)}$であれば10のビットの繰り返し、$8000_{(16)}$であれば最上位ビット(ビット番号15)のみが1であとは全て0であることが直ちに分かります。

このように、慣れてくると16進数を見ただけで、その数値のビットパターンを連想することができます。

(*1) 16進数字A〜Fに対して、小文字のa〜fを使うこともあります。しかし、CASL IIは大文字しか扱わないので、本書では、A〜Fのみを使うことにします。

表 2-6　代表的な16ビット数値

| 16進数 | 2進数 |
|---|---|
| 0000 | 0000000000000000 |
| 0001 | 0000000000000001 |
| FFFF | 1111111111111111 |
| F000 | 1111000000000000 |
| 8000 | 1000000000000000 |
| 8888 | 1000100010001000 |
| 1111 | 0001000100010001 |
| AAAA | 1010101010101010 |

表 2-7　10進数、8進数、16進数の表

| 10進 | 8進 | 16進 | 10進 | 8進 | 16進 | 10進 | 8進 | 16進 | 10進 | 8進 | 16進 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 16 | 20 | 10 | 32 | 40 | 20 | 48 | 60 | 30 |
| 1 | 1 | 1 | 17 | 21 | 11 | 33 | 41 | 21 | 49 | 61 | 31 |
| 2 | 2 | 2 | 18 | 22 | 12 | 34 | 42 | 22 | 50 | 62 | 32 |
| 3 | 3 | 3 | 19 | 23 | 13 | 35 | 43 | 23 | 51 | 63 | 33 |
| 4 | 4 | 4 | 20 | 24 | 14 | 36 | 44 | 24 | 52 | 64 | 34 |
| 5 | 5 | 5 | 21 | 25 | 15 | 37 | 45 | 25 | 53 | 65 | 35 |
| 6 | 6 | 6 | 22 | 26 | 16 | 38 | 46 | 26 | 54 | 66 | 36 |
| 7 | 7 | 7 | 23 | 27 | 17 | 39 | 47 | 27 | 55 | 67 | 37 |
| 8 | 10 | 8 | 24 | 30 | 18 | 40 | 50 | 28 | 56 | 70 | 38 |
| 9 | 11 | 9 | 25 | 31 | 19 | 41 | 51 | 29 | 57 | 71 | 39 |
| 10 | 12 | A | 26 | 32 | 1A | 42 | 52 | 2A | 58 | 72 | 3A |
| 11 | 13 | B | 27 | 33 | 1B | 43 | 53 | 2B | 59 | 73 | 3B |
| 12 | 14 | C | 28 | 34 | 1C | 44 | 54 | 2C | 60 | 74 | 3C |
| 13 | 15 | D | 29 | 35 | 1D | 45 | 55 | 2D | 61 | 75 | 3D |
| 14 | 16 | E | 30 | 36 | 1E | 46 | 56 | 2E | 62 | 76 | 3E |
| 15 | 17 | F | 31 | 37 | 1F | 47 | 57 | 2F | 63 | 77 | 3F |

## 演習問題2-5

次の16進数を2進数に変換してください。

(1) $5CCA_{(16)}$
(2) $FDDA_{(16)}$
(3) $1234_{(16)}$
(4) $4444_{(16)}$
(5) $CAFE_{(16)}$

## 2.6 負のデータの扱い

### 2.6.1 負のデータと2の補数表現

16ビットのデータは、$2^{16}$通りの表現が可能ですから、65536通りのデータを表現することができます。これを10進数の数値に対応させると、0から65535までの数値を表現することができます。

しかし、数値には負の数も存在します。COMET IIでは、負の数を表現するのに、**2の補数**という方法を使います。

**ポイント**

COMET IIでは、負の数を表現するのに、2の補数を用いる。

16ビットで説明すると、桁が多くなって説明するのが大変なので、とりあえず3ビットで説明します。次のようなビット表現を想定します。

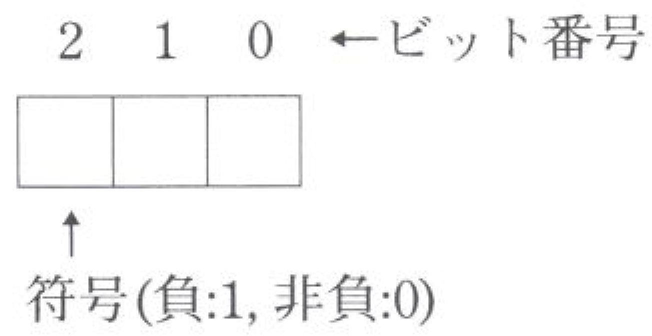



**図2-3 3ビットの構成**

3ビットのデータは$2^3$通りの表現が可能ですから、8通りのデータを表現することができます。10進数値に対応させると、3ビットでは0から7までの数値を表現することができます。しかし、0から7までの数値は、すべて0以上の数値です。これは、ちょうど表2-8の符号無し表現の部分に相当します。

**表 2-8 3ビットのビットパターンとそれが表現する数**

| 2進数 | 符号無し表現 | 符号付き表現 |
|---|---|---|
| 000 | 0 | 0 |
| 001 | 1 | 1 |
| 010 | 2 | 2 |
| 011 | 3 | 3 |
| 100 | 4 | −4 |
| 101 | 5 | −3 |
| 110 | 6 | −2 |
| 111 | 7 | −1 |

一般にコンピュータでは、2進数のビットパターンの数値の扱いを、符号無しデータと符号付きデータの2通りの仕方で処理します。ここで大切なことは、同じビットパターンのデータに対してコンピュータが２つの異なる解釈をするということです。

符号無しデータでは、2進のビットパターン($b_2\ b_1\ b_0$)をそのまま、$b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$という

形で表現します。しかし、符号無しデータの場合、最上位ビット(3ビットの場合には、ビット番号2、16ビットの場合にはビット番号15)を符号ビットとみなして、最上位ビットに1が立っているとその数値を負の数とみなします。表2-8では、ビット番号2が1となっている数は、符号付き表現では負の数となっています。

**ポイント**

符号付きデータでは、最上位ビット(ビット番号15)を符号ビットとみなし、それが1ならば負、0ならば非負のデータとしている。

2の補数という方法で負の数を表現する場合、3ビットでは$-2^2$から$2^2-1$までの数が表現できます。$100_{(2)}$が一番小さな値$(-4)$になり、$111_{(2)}$が$-1$になります。

一般に、nビットの場合$-2^{n-1}$〜$2^{n-1}-1$までの数を表現でき、$100...00_{(2)}$(全部でnビット)が$-2^{n-1}$を、$111...11_{(2)}$(全部でnビット)が$-1$を表現します。

COMET II の数値は16ビットなので、$-2^{15}$〜$2^{15}-1$、すなわち、$-32768$〜$32767$までを表現でき、$1000000000000000_{(2)}$が$-32768$となり、$1111111111111111_{(2)}$が$-1$となります。

**表 2-9 16ビットのビットパターンとそれが表現する数**

| 2進数 | 符号無し表現 | 符号付き表現 |
|---|---|---|
| 0000000000000000 | 0 | 0 |
| 0000000000000001 | 1 | 1 |
| 0000000000000010 | 2 | 2 |
| 0000000000000011 | 3 | 3 |
| 〜 | 〜 | 〜 |
| 0111111111111111 | 32767 | 32767 |
| 1000000000000000 | 32768 | −32768 |
| 1000000000000001 | 32769 | −32767 |
| 〜 | 〜 | 〜 |
| 1111111111111100 | 65532 | −4 |
| 1111111111111101 | 65533 | −3 |
| 1111111111111110 | 65534 | −2 |
| 1111111111111111 | 65535 | −1 |

さて、ここで大切なことは、前述したように、コンピュータの内部では、1つのビットパターンに対して、符号付きデータと符号無しデータの2つのデータというそれぞれ異なる2つの解釈があるということです。2進数$1000000000000001_{(2)}$は、符号無しデータとみなした場合32769ですが、符号付きデータとみなした場合には、$-32767$となります。COMET II の各命令は、データを符号付きデータとみなして計算するものと符号無しデータとみなして計算するものとに分けられています。たとえば、データの加算に対して、符号付きデータで計算する場合にはADDAという命令を用いますが、符号無しデータで計算する場合にはADDLという命令を用います。

## ポイント

符号付きデータを加算する時はADDA命令、符号無しデータを加算する時はADDL命令を用いる

5.4節「フラグレジスタと条件判断」で、詳しく説明しますが、COMET IIには、演算結果が負の値になった場合、「演算結果が負になりましたよ」という印が付くサインフラグ(SF)というものがあります。

ADDAを用いて、データを符号付きデータであるとみなして計算した結果が1000000000000001(2)となった場合には、−32767と見なされるので、SFには「負になった」という意味の1がセットされ、ADDLを用いてデータを符号付無しデータとみなして計算した結果が1000000000000001(2)となった場合には、32769と見なされるので、SFには「非負になった」という意味の0がセットされます。

## 演習問題2-6-1

次の4ビットの2進数を10進数の符号無し表現と符号付き表現(ビット番号3を符号ビットとする)で示してください。

(1) 1111
(2) 1010
(3) 1110
(4) 0001
(5) 0101

### 2.6.2 負の数(2の補数)への変換

さて、負の数の表現方法は、前項で説明したので理解できたと思います。そこで、本項では、正の数を負の数に変換したり、その逆を行う方法について説明します。

ある数の2の補数（いわゆる、負の数）を求めるには、次のようにします。たとえば、3の2の補数を求めて、−3にしてみましょう。

その手順は、とても単純で、次の2つの手順を行うだけです。

(1) すべてのビットを反転する
(2) 1を加える

3の場合には、次のようになります。

```
 3   0000000000000011
          ↓(ビット反転)
     1111111111111100
 ＋                 1   ←1を加える
─────────────────────
－3  1111111111111101
```

ビット反転して1を加えたところ、1111111111111101 (2)が求められました。表2-9をみると、この数値が－3になっていることが確認できますね。それでは、この数値に対してまったく同じ操作をしてみましょう。

```
－3  1111111111111101
          ↓(ビット反転)
     0000000000000010
 ＋                 1   ←1を加える
─────────────────────
 3   0000000000000011
```

なんと、ちゃんと3に戻っていますね。つまり、2の補数を求めるというのは、符号を反転させることに等しく、また、このようにとても単純な操作で出来てしまうのです。

### ポイント

2の補数を求める手順

(1) すべてのビットを反転する

(2) 1を加える

### 演習問題2-6-2

次の数値の2の補数を16ビットの2進数で求めなさい。

(1) 5

(2) 10

(3) －5

### 2.6.3　2の補数と減算の仕組み

COMET IIには減算の命令として、SUBA(符号付きデータ)とSUBL(符号無しデータ)の2つがあります。しかし、整数の減算を行うという演算回路は実際には用意されていないこともあるのです。それは、減算という操作が、2の補数を加えるという操作でできてしまうからなのです。

11 － 6 という計算について調べてみましょう。この 2 つの数値は16ビットの2進数では次の通りで

す。

```
11 : 0000000000001011
 6 : 0000000000000110
```

さて、6の2の補数を求めて、−6に変換して、これを11に加えてみましょう。まず、6の2の補数を求めます。

```
 6   0000000000000110
       ↓(ビット反転)
     1111111111111001
+                   1   ←1を加える
−6   1111111111111010
```

−6は、1111111111111010(2)となりました。11とこの値を加えてみましょう。次の例のように答えは17ビットになりますが、COMET IIには格納する部分が16ビット分しかありませんので、オーバーフローした部分は切り捨てられます。

```
   11 : 0000000000001011
 + −6 : 1111111111111010
        10000000000000101
        ↓
     切り捨て       ↓
         0000000000000101
```

答えは、101(2)、すなわち5になりました。これは、11−6の答えです。つまり、2の補数を使えば、負の数を、そのまま加算することによって、正しく演算が行われるのです。

それでは、2−6を計算してみましょう。−6は1111111111111010(2)であることが分かっていますから、次の計算を行えば良いことが分かります。

```
   2 : 0000000000000010
 +−6 : 1111111111111010
       1111111111111100
```

今回の演算では、オーバーフローした部分はありません。この値は、−4であり2−6が計算されたことがわかります。

## 演習問題2-6-3

次の計算を16桁の2進数のレベルで2の補数を用いて行いなさい。

(1) 5−3
(2) 2−10
(3) 14−5

## 2.7 各進数表現の変換

実際にアセンブラ言語でプログラムを作成する際には、10進数を2進数や16進数に変換したり、あるいはその逆の変換を行う必要があります。ここでは、その方法について説明したいと思います。

### 2.7.1 10進数のn進数への変換

10進数をn進数に変換するには、その数をnで割って余りを並べていきます。

(1) 10進数→2進数

10進数dを2進数に変換することを考えてみましょう。

d=242を2進数に変換する場合、次のように2で割って余りを並べていきます。

```
242 ÷ 2 =121  ... (余り) 0   ↑
121 ÷ 2 = 60  ...        1   |
 60 ÷ 2 = 30  ...        0   |
 30 ÷ 2 = 15  ...        0   |
 15 ÷ 2 =  7  ...        1   |
  7 ÷ 2 =  3  ..         1   |
  3 ÷ 2 =  1  ...        1   |
  1 ÷ 2 =  0  ...        1   | 並べる順序
```

そして、余りを逆順に並べると、$242_{(10)} = 11110010_{(2)}$となります。

この仕組みについて説明しましょう。実は、$11110010_{(2)}=242_{(10)}$という数を分析すると、次のことがわかります。

$11110010_{(2)}$を2で割ることを考えます。この数は定義によれば、$1\times2^7+1\times2^6+1\times2^5+1\times2^4+0\times2^3+0\times2^2+1\times2^1+0\times2^0$でした。これを2で割ると、$1\times2^6+1\times2^5+1\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$となります。

これは、$1111001_{(2)}=121_{(10)}$ですね。そして、最下位桁は、元の数を2で割った時の余りになります(偶数のとき0、奇数のとき1)。

つまり次のことが分かります。

```
                    242を2で割った時の余り
                              ↓
1   1   1   1   0   0   1   0 ──────→ 0が求められた
←  242を2で割った値  →
  (すなわち、121)
```

これで、最下位桁が求められました。さらに$242_{(10)}$を2で割った値$1111001_{(2)}=121_{(10)}$について同じ

処理をします。

```
                121を2で割った時の余り
                          ↓
1  1  1  1  0  0  1 ──────────→ 1が求められた
←121を2で割った値→
  (すなわち、60)
```

これで、2桁目が求められました。この手順を繰り返せば、すべての2進数が求められます。

## ポイント

10進数を2進数にするには、2で割って余りを逆順に並べる

## 演習問題2-7-1(1)

次の10進数を2進数に変換しなさい。

(1) 159
(2) 2152
(3) 10000

(2) 10進数→8進数

10進数を8進数に変換する手順も同じで、8で割って余りを求めます。

```
242 ÷ 8 = 30   ...(余り) 2   ↑
 30 ÷ 8 =  3   ...(余り) 6   │
  3 ÷ 8 =  0   ...(余り)3    │ 並べる順序
```

余りを逆順に並べて、 $242_{(10)} = 362_{(8)}$となります。

この仕組みについて説明しましょう。実は、$362_{(8)}=242_{(10)}$という数を分析すると、次のことがわかります。

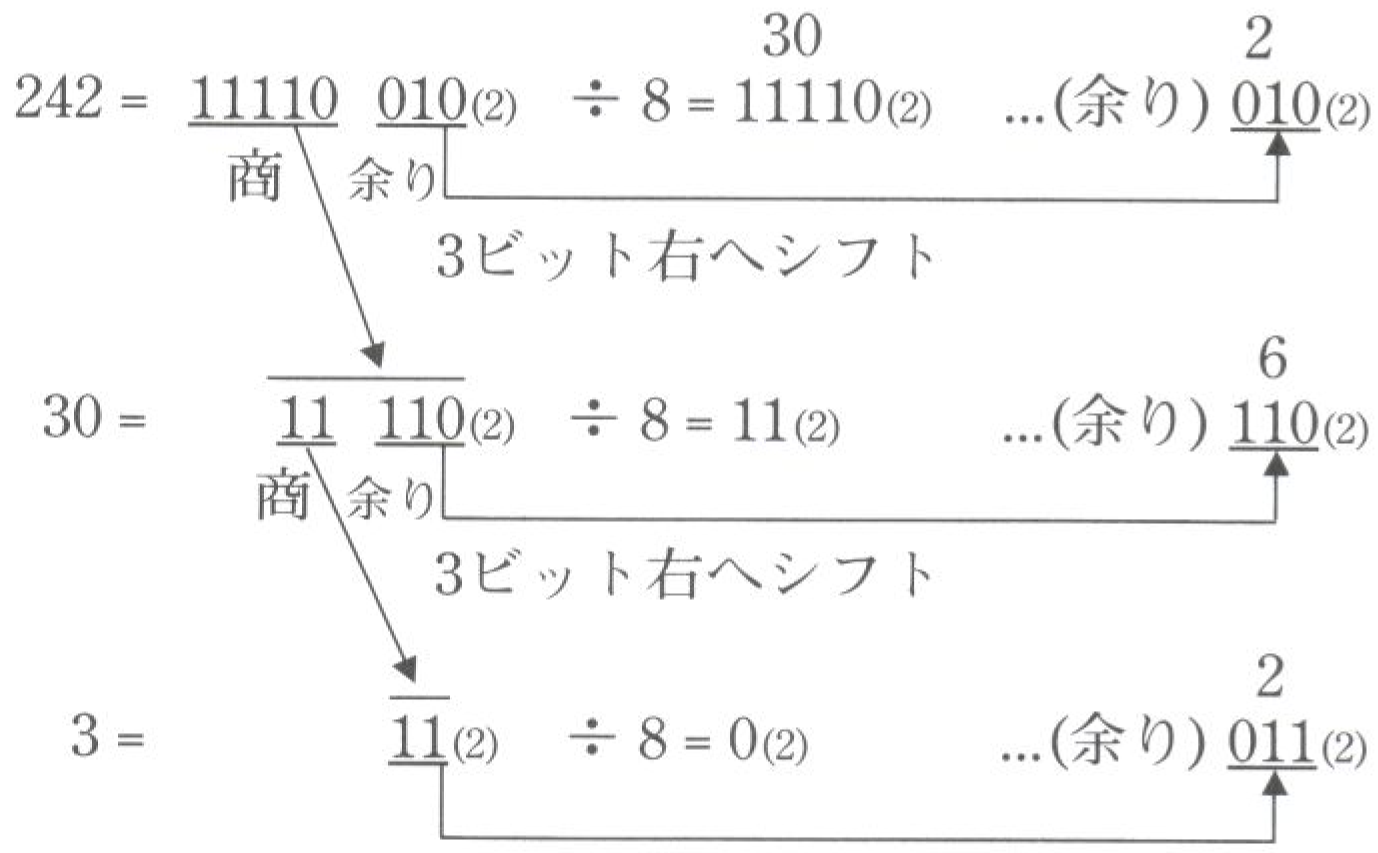
$362_{(8)}$を8で割ることを考えます。これは定義によれば、$3\times8^2+6\times8^1+2\times8^0$でした。これを8で割ると、$3\times8^1+6\times8^0$となります。これは、$36_{(8)}=30_{(10)}$で、$362_{(8)}$を8で割った値です。そして、$362_{(8)}$の

最下位桁である2(8)は、元の数362(8)を8で割った時の余りになります。

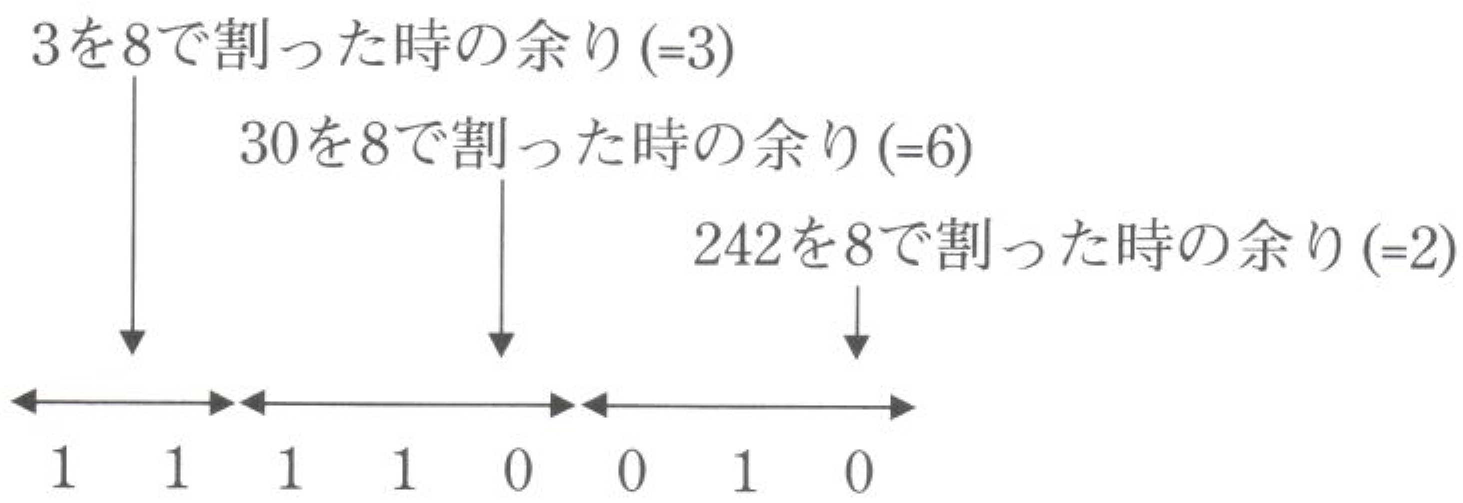
つまり、362という8進数の各桁は次の関係になっているのです。




これで、最下位桁が求められました。ところで、これを2進数で見ると次のようになります。2進数を8で割るということは、その数値の各桁を3つ右へ移動させるのと同じ事です。この操作をシフト演算と呼びます。シフト演算については6.5節「シフト演算命令」で詳しく説明します。




ビット全体は、次のようになります。




実は、これは2進数を3ビットづつ区切るとなぜ8進数になるかの説明になっています。要するに、各桁は8で割った余りになっているので、このような単純な対応づけができるのです。

**ポイント**

10進数を8進数にするには、8で割って余りを逆順に並べる

## 演習問題2-7-1 (2)

次の10進数を8進数に変換しなさい。

(1) 159
(2) 2152
(3) 10000

(3) 10進数→16進数

10進数を16進数に変換するには、16で割って余りを逆順に並べます。
余りが10〜15のときは、A〜Fの数字に対応させます。

```
242 ÷ 16 = 15 …  (余り) 2
 15 ÷ 16 =  0 …        15 → F
```

余りを逆順に並べて、$242_{(10)} = F2_{(16)}$となります。

## ポイント

10進数を16進数にするには、16で割って余りを逆順に並べる

## 演習問題2-7-1 (3)

次の10進数を16進数に変換しなさい。

(1) 159
(2) 2152
(3) 10000

### 2.7.2 2進数の8進数、16進数への変換

2進数が求められている場合、それを用いて8進数、16進数を簡単に求めることができます。

2.4節、2.5節で説明したように、$10110_{(2)}$を8進数に変換する場合には、3ビットごとに分け10 110とし（3桁に満たない場合には0を補います）、16進数に変換する場合には、4ビットごとに分け1 0110とします（4桁に満たない場合は0を補います）。

| 010 | 110 | 0001 | 0110 |
|---|---|---|---|
| 2 | 6 | 1 | 6 |

左の図から$10110_{(2)}$は$26_{(8)}$, $16_{(16)}$であることがわかります。

## ポイント

2進数を8進数にするには、3桁ごとに区切る
2進数を16進数にするには、4桁ごとに区切る

## 演習問題2-7-2

次の2進数を8進数、16進数に変換しなさい。

(1) $1011101_{(2)}$
(2) $11110000111100001111_{(2)}$
(3) $10101101011010101_{(2)}$

### 2.7.3 8進数値、16進数値から2進数値への変換

8進数値、16進数値から2進数値への変換もまた、とても簡単です。各桁をそれに対応する2進数に変換すればよいのです。

$123_{(8)}$と$123_{(16)}$は次のように変換されます。

| 1 | 2 | 3 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| 001 | 010 | 011 | 0001 | 0010 | 0011 |

したがって、$123_{(8)}=1010011_{(2)}$、$123_{(16)}=100100011_{(2)}$となります。

## ポイント

8進数、16進数を2進数にするには、各桁を対応する2進数に変換する

## 演習問題2-7-3

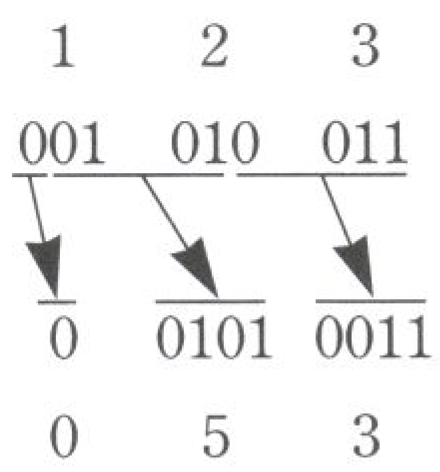
次の数を2進数に変換しなさい。

(1) 156(8)

(2) FFAB(16)

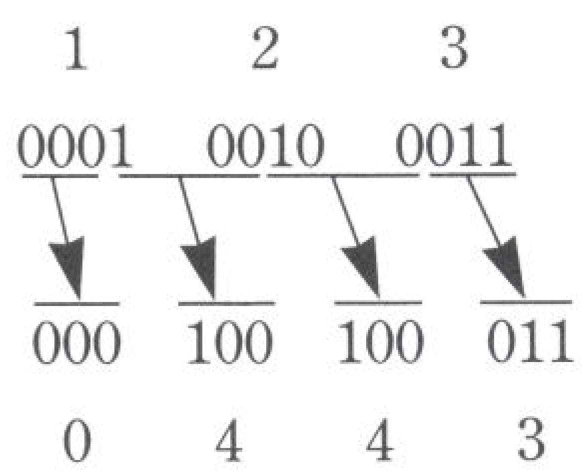
### 2.7.4 8進数・16進数の相互変換

8進数値から16進数値、あるいはその逆の変換は、いったん2進数に変換してから行えばよいでしょう。たとえば、123(8)を16進数に変換するには




となり、123 (8)=53 (16)となることがわかります。

また、16進数の123 (16)を8進数に変換するには




となり、123 (16)=443 (8)であることが分かります。

## ポイント

8進数、16進数の相互変換は、2進数を介する

## 演習問題2-7-4

次の8進数を16進数に、16進数を8進数に変換しなさい。

(1) $1234_{(8)}$
(2) $ABCD_{(16)}$
(3) $7777_{(8)}$

### 2.7.5 n進数の10進数への変換

n進数 $m_5\,m_4\,m_3\,m_2\,m_1\,m_0$ は、10進数では次のように表現できます。

$$m_5 \times n^5 + m_4 \times n^4 + m_3 \times n^3 + m_2 \times n^2 + m_1 \times n^1 + m_0 \times n^0$$

2進数、8進数、16進数の場合には、次のように表現できます。

2進数： $m_5 \times 2^5 + m_4 \times 2^4 + m_3 \times 2^3 + m_2 \times 2^2 + m_1 \times 2^1 + m_0 \times 2^0$
8進数： $m_5 \times 8^5 + m_4 \times 8^4 + m_3 \times 8^3 + m_2 \times 8^2 + m_1 \times 8^1 + m_0 \times 8^0$
16進数： $m_5 \times 16^5 + m_4 \times 16^4 + m_3 \times 16^3 + m_2 \times 16^2 + m_1 \times 16^1 + m_0 \times 16^0$

つまり、各桁のこの重み付けを計算すればよいのです。

たとえば、$10110_{(2)}$, $245_{(8)}$, $3EA2_{(16)}$は次のようになります。

$$10110_{(2)} = 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0 = 1\times16 + 0\times8 + 1\times4 + 1\times2 + 0\times1$$
$$= 16 + 4 + 2 = 22$$

$$245_{(8)} = 2 \times 8^2 + 4 \times 8^1 + 5 \times 8^0 = 2\times64 + 4\times8 + 5\times1 = 128 + 32 + 5 = 165$$

$$3EA2_{(16)} = 3 \times 16^3 + E \times 16^2 + A \times 16^1 + 2 \times 16^0$$
$$= 3\times4096 + 14\times256 + 10\times16 + 2\times1 = 12288 + 3584 + 160 + 2 = 16034$$

16進数の場合には、A, B, C, D, E, Fの各数字を使うので、これらは、それぞれ10, 11, 12, 13, 14, 15の値に変換して計算します。また、実際に計算をする場合には、上記のように数式の形で書かないで、重み付けの部分だけを書いた方が効率が良いでしょう。

その方法で、$10110_{(2)}$, $245_{(8)}$, $3EA2_{(16)}$を10進数に変換する場合には、次のようにします。

2進数の場合

| … | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | 1 | 0 | 1 | 1 | 0 |

16 ＋ 4＋2 = **22**

8進数の場合

| … | 32768 | 4096 | 512 | 64 | 8 | 1 |
|---|---|---|---|---|---|---|
| | | | | 2 | 4 | 5 |

128＋ 32 ＋ 5 = **165**

16進数の場合

| … | 1048576 | 65536 | 4096 | 256 | 16 | 1 |
|---|---|---|---|---|---|---|
| | | | 3 | E | A | 2 |
| | | | | 14 | 10 | |

12288＋ 3584＋ 160 ＋ 2 = **16034**

## ポイント

2進数、8進数、16進数を10進数にするには、各桁の重みを合算する。

## 演習問題2-7-5

次の数を10進数に変換しなさい。

(1) $110101011_{(2)}$

(2) $12345_{(8)}$

(3) $FC_{(16)}$

# 総合問題2

**[演習2-1]** 次の10進数値を、指定された進法に変換しなさい。

(a) 2進数
(1) 5　(2) 10　(3) 45　(4) 650　(5) 1025

(b) 8進数
(1) 7　(2) 125　(3) 3333　(4) 10123　(5) 100000

(c) 16進数
(1) 100　(2) 2345　(3) 1025　(4) 64000　(5) 100000

**[演習2-2]** 次の数値を10進数に変換しなさい。

(1) $1011010101_{(2)}$　(2) $726_{(8)}$　(3) $37337261_{(8)}$
(4) $37337261_{(16)}$　(5) $\mathrm{CAFE}_{(16)}$　(6) $\mathrm{12BC34F}_{(16)}$

**[演習2-3]** 次の数値に1を加えた数をそれと同じ進法で答えなさい。

(1) $1011011_{(2)}$　(2) $101011000_{(2)}$　(3) $372177_{(8)}$　(4) $323712_{(8)}$
(5) $\mathrm{AB012}_{(16)}$　(6) $\mathrm{FD01FFE}_{(16)}$　(7) $\mathrm{12ABC}_{(16)}$　(8) $\mathrm{FFFFFF}_{(16)}$

**[演習2-4]** 次の2進法の加算を行いなさい。

(1) $110101_{(2)}+10111_{(2)}$　(2) $11111_{(2)}+100_{(2)}$　(3) $10110101_{(2)}+110110_{(2)}$

**[演習2-5]**

10進数80000を2進数で表現し、その下位16ビットだけを抜き取ったとき、その値はいくつですか？

# 第3章 アセンブラ言語によるプログラミングのための前提知識

## 3.1 COMET II のハードウェア構成

### 3.1.1 COMET II の主記憶装置

**主記憶装置**は、そのコンピュータで扱うプログラムやデータを一時的に格納しておくための電子回路です。プログラムやデータは、通常は、ハードディスクなどの補助記憶装置の中に格納されています。コンピュータで必要になったときに、これらが入力装置を介して主記憶装置の中に転送されます。転送されるといっても、補助記憶装置上にはもとのプログラムやデータがそのままの形で残っています。したがって、「転送」というよりは「複写」という方が適切です。

主記憶装置は、通常**ULSI**(Ultra Large Scale IC:超々集積回路(*1)であり、この中に電子的にデータが記憶されます。そのため、電源を切ると記憶されたデータは無くなってしまうので、注意が必要です。

さて、このようにプログラムやデータを格納する主記憶装置ですが、通常は、**1語** (word)あるいは**1ワード**という単位でデータの出し入れが行われます。1語の大きさは、コンピュータによって異なります。一般のパソコンで用いられているPentiumは、1語の大きさは32ビットですが、COMET IIでは、1語の大きさは16ビットです。したがって、COMET II には、符号付きで−32768〜32767、符号無しで0〜65535の整数のデータを格納することができます。

COMET II の主記憶装置には、65536個の語が用意されています。これらの語には、0番から65535番まで連続した番地が付いています。これを**アドレス**(address)と呼びます。COMET II の場合、0番地から65535番地までのアドレスが存在し、ここにデータを出し入れするということになります。0から65535の値を符号無しデータとして表現するには、16ビットが必要となります。従って、COMET II のアドレスを表現するには16ビット（1語）が必要となります。

COMET IIの主記憶装置にある65536個のどの語に対してもデータを出し入れすることができます。しかし、一般のコンピュータは、オペレーティング・システムなど他のプログラムが使用中の語(メモリ)にはアクセス(データの出し入れ)できないようになっています。

(＊1) 集積回路(IC: Integrated Circuit)の中に1万個以上の素子が入っているものをLSI, 10万個以上の素子が入っているものをVLSI(Very Large Scale IC)、100万個以上の素子が入っているものをULSIと呼びます。

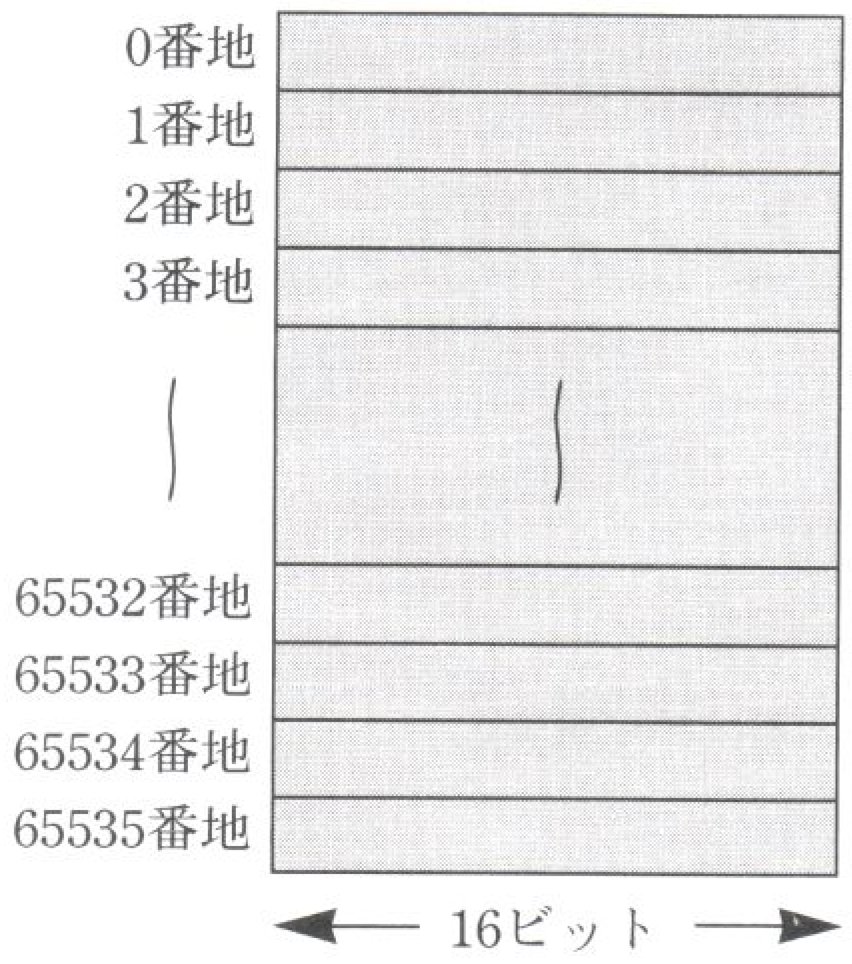



**図3-1 COMET IIの主記憶装置**

データを主記憶装置に出し入れする際には、以下に示すようにアドレスを指定して行います。

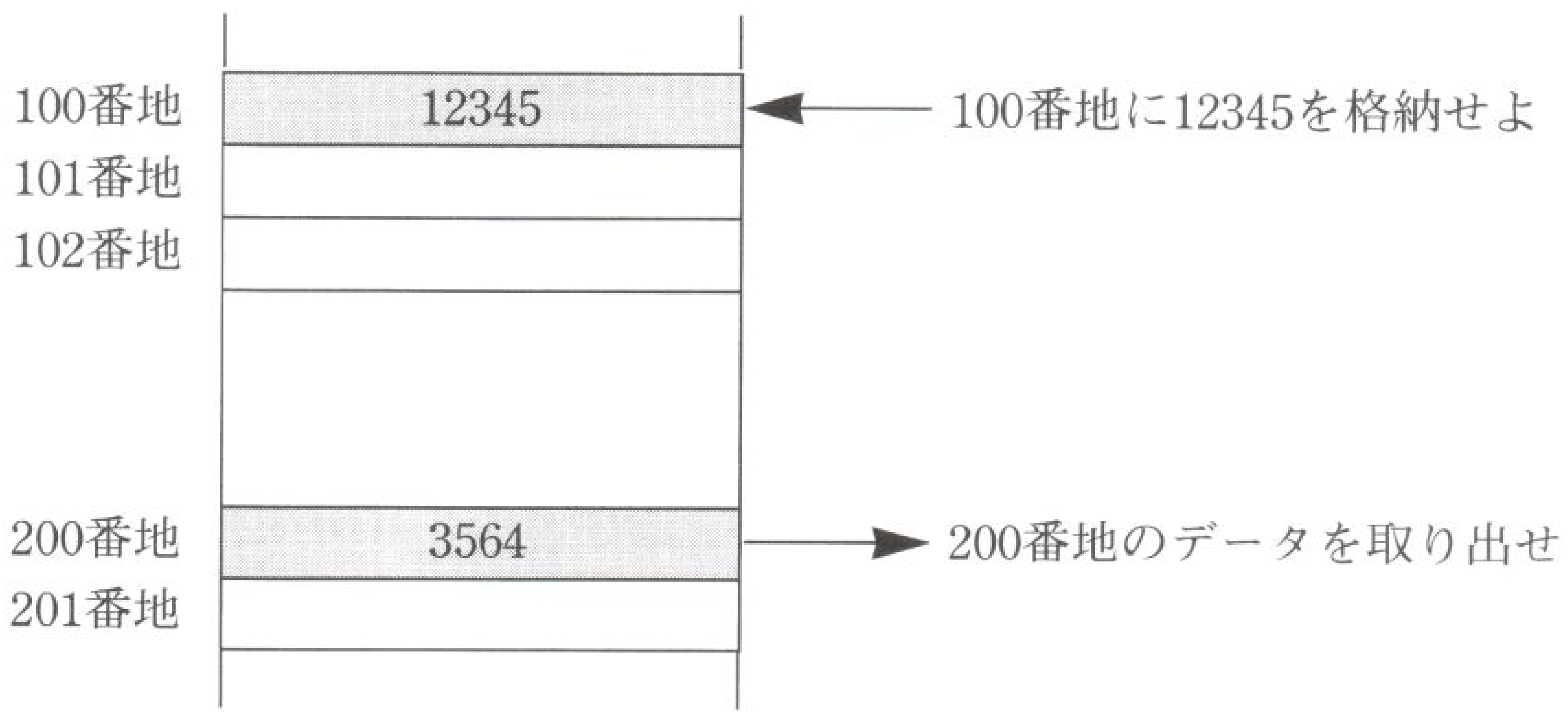



**図3-2 主記憶装置からのデータの格納と取り出し**

**ポイント**

COMET IIの主記憶装置には16ビットの語が65536個用意されている。
先頭から0番地、1番地、...、65535番地のアドレスが付いている。

### 3.1.2 COMET II の入出力装置

一般のコンピュータには、さまざまな**入出力装置**が付いています。しかし、単純化されたコンピュータであるCOMET II には、非常に簡単な入出力装置しか付いていません。COMET IIの仕様では、入出力装置についての具体的な定義が示されていません。しかし、入力装置としては、キーボードあるいは補助記憶装置(上のファイル)などが仮定できるでしょう。また、出力装置としては、ディスプレイあるいは補助記憶装置(上のファイル)などが仮定できるでしょう。使用する入出力装置は、プログラムの実行に先立って、オペレーティングシステムの機能を用いて決定することになっています。

どのような装置からデータを入出力するにせよ、COMET IIでは、指定された特定番地から何語か

を連続して入出力するようになっています。これについては、4.6「マクロ命令」で説明します。

### 3.1.3 COMET II の演算装置・制御装置

COMET II の制御装置には、図3-3に示すように、**レジスタ**(register)と呼ばれる小さなメモリ（記憶装置）があります。レジスタとは、プログラムの動作状況を記憶したり、一時的にデータを記憶するために用いられます。

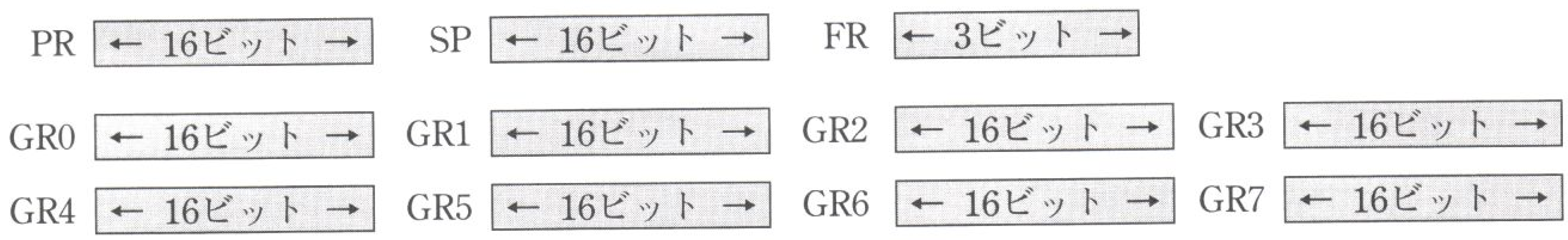



**図 3-3 COMET IIのレジスタ構成**

COMET II のレジスタには、図3-3に示すようにプログラムレジスタ、スタックポインタ、汎用レジスタ(8個)、フラグレジスタの4種類があります。

**プログラムレジスタ**は、**PR**(Program Register)で表現され、次に実行すべき機械語の先頭アドレスが格納されています。アドレスを保持するのでPRの寸法（サイズ）は16ビットです。

**スタックポインタ**は、**SP**(Stack Pointer)で表現され、主記憶装置内のスタックと呼ばれる特定部分の先頭アドレスを保持しています。SPもアドレスを保持するので、その寸法は16ビットです。

**汎用レジスタ**(General Register)は、**GR0, GR1, GR2, GR3, GR4, GR5, GR6, GR7** の8個があり、演算のためのデータを格納します。実際の計算は、汎用レジスタ同士あるいは汎用レジスタと主記憶装置の間で行われ、結果は汎用レジスタに格納されます。これらの汎用レジスタは計算をするだけなら、基本的にどれも同じように使えます。しかし、後述する指標レジスタとして用いる場合には、GR1～GR7だけが使えて、GR0は使えません。GR0だけはちょっと特殊だぞと思っていてください。

さて、実際の計算は、汎用レジスタを用いて行われます。たとえば、図3-4(a)に示すように、主記憶装置の3番地に12345、4番地に25が格納されているとします。このとき、LD(ロード)(詳細は後述)という命令によって、汎用レジスタに主記憶装置の内容を複写できます。図3-4(a)では、3番地の内容をGR5に複写しています。このことを**ロード**(Load)といいます。

そして、算術加算(ADDA：詳細は後述)という命令によって、図3-4(b)に示すように、他の汎用レジスタの内容(この例では、GR2)とGR5の内容を加えて、その結果をGR5に戻します。

あるいは、図3-4(c)に示すように、主記憶装置の4番地の内容とGR5の内容を加えて、その結果をGR5に戻すこともできます。このように、演算は、汎用レジスタ間、あるいは汎用レジスタと主記憶装置の間で行われるということを覚えておいてください。

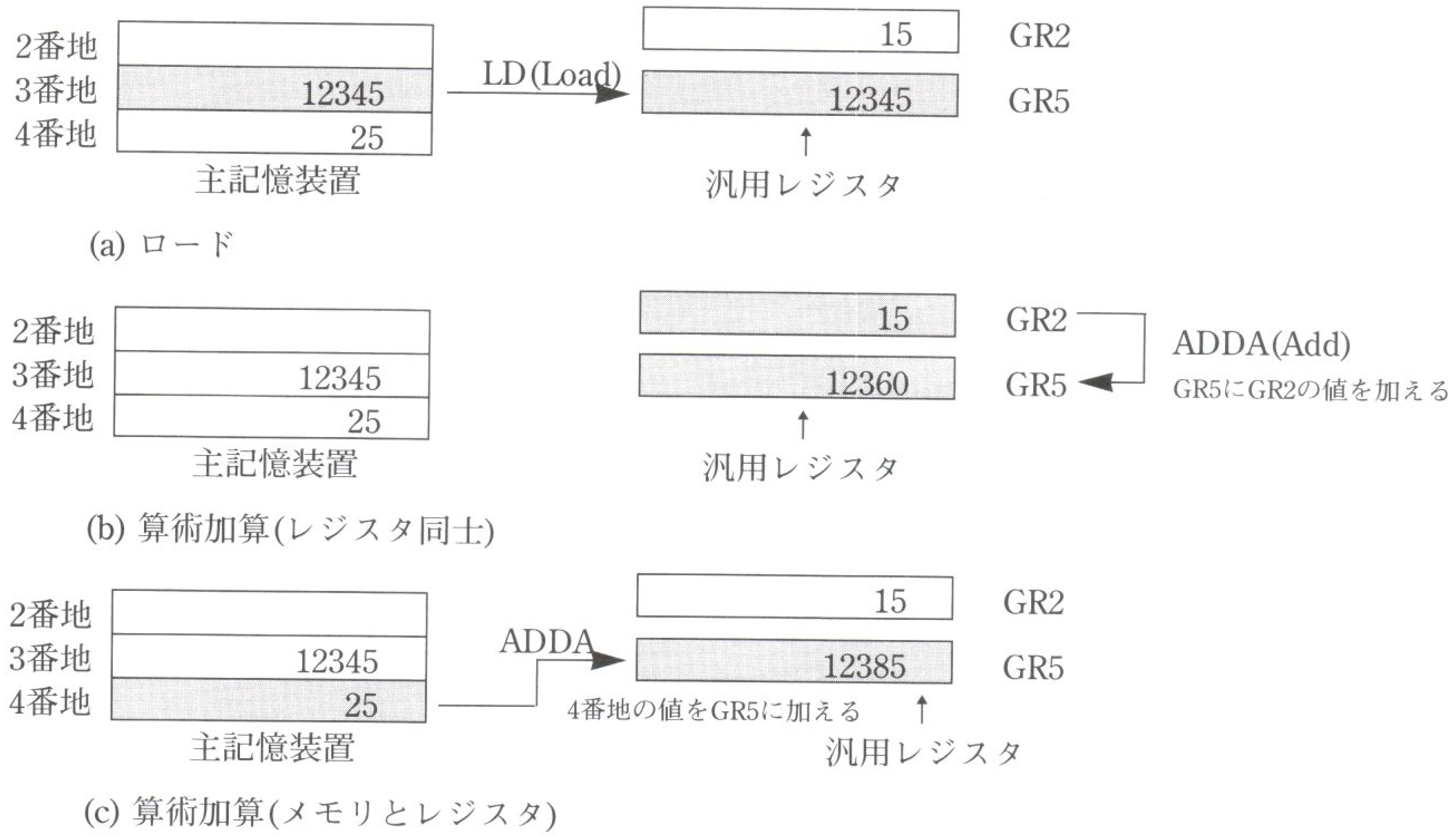



**図3-4　データのロードと算術計算**

**フラグレジスタ**は、**FR**(Flag Register)で表現され、プログラムの実行状態を示します。FRだけは、他のレジスタと異なり、3ビットしかありません。この3つのビットは、それぞれ、OF(Overflow Flag)、SF(Sign Flag)、ZF(Zero Flag)と呼ばれます。たとえば、SFは演算結果が負になった場合に1となり、そうでない(すなわち、0以上)場合には0となります。この結果を参照して、プログラムの動作を変えることができますが、詳細は6.6節「分岐命令」で示します。

## ポイント

COMET II には、プログラムレジスタ(PR)、スタックポインタ(SP)、汎用レジスタ(8個:GR0,GR1,GR2,GR3,GR4,GR5,GR6,GR7)、フラグレジスタ(FR)の4種類があります。
PR, SP, GR0〜GR7は16ビット構成。FRは、3ビット(OF, SF, ZF)構成。

## 演習問題3-1

(1) 各汎用レジスタで表現できる数の大きさはいくつですか。

(2) COMET IIには、0番地から65535番地までの語がありますが、PRで表現できない番地はありますか。

## 3.2 COMET II の機械語

### 3.2.1 機械語とは

現在のコンピュータは、びっくりするような映像を作り出したり、面白いゲームで人間の相手をしてくれたり、本当にいろいろなことができます。しかし、コンピュータは、実際には、機械語と呼ばれる非常に単純な命令を、淡々と実行しているに過ぎないのです。この単純な命令とはたとえば、次のようなものです。

主記憶装置8000番地の内容を汎用レジスタのGR4に格納せよ。
GR4の内容とGR3の内容を加えて、GR4に格納せよ。
GR4の内容を主記憶装置の8001番地に格納せよ。

この個々の命令を**機械語**と呼びます。上記の命令は、人間の言葉に直して書いていますが、実際には、どのようなものなのでしょう。残念ながら、COMET IIの仕様では、具体的な機械語の詳細は定義されていませんが、COMET IIの参考資料(＊1)に、それが示されています。本書、および本書で提供するCASLDV(COMET IIエミュレータ、CASL IIアセンブラ)も、この参考資料に完全に準拠しています。

それによると、上記の機械語は次のようになります。主記憶装置1語が16ビットですから、1行に1語ずつ示しています。

```
0001000001000000
0001111101000000
0010010001000011
0001000101000000
0001111101000001
```

上記の表記は、2進数で記述しています。電流のON/OFFの様子は良く分かるのですが、どのような数値であるのかがよく分かりません。そこで16進数で表示すると次のようになります。

```
1040
1F40
2443
1140
1F41
```

16桁から4桁になったので、多少すっきりしていますが、残念ながらこの意味を直ちに理解することは困難です。この機械語の意味は、次のようになります。

1040 汎用レジスタのGR4に次の行に示すアドレスの内容を格納せよ。
1F40 主記憶装置8000番地(1F40(16)は10進数では、8000)
2443 GR4の内容とGR3の加えて、GR4に格納せよ。
1140 汎用レジスタのGR4の内容を次の行に示すアドレスに格納せよ。
1F41 主記憶装置8001番地(1F41(16)は10進数では、8001)

(＊1) 付録参照。

人間にとって、まったく理解しがたい機械語ですが、コンピュータにとっては、この方が都合が良いのです。

### 3.2.2 COMET II の機械語の概要

それでは、COMET IIには、どのような機械語があるのでしょうか。まず、その一部を次に示します。表3-1の中で、wおよびzは0から7までのどれかの数値であるとします。

表3-1 COMET IIの機械語(一部)

| 機械語(16進数) | 意味 |
|---|---|
| 14wz | レジスタ GRz の内容をレジスタ GRw にロード(複写)する |
| 24wz | レジスタ GRz の内容をレジスタ GRw に加算する。 |
| 25wz | レジスタ GRz の内容をレジスタ GRw から減算する。 |

たとえば、「レジスタ GR3 の内容をレジスタ GR7 にロード(複写)する」機械語は$1473_{(16)}$になり、「レジスタ GR5 の内容をレジスタ GR0 に加算する」機械語は$2405_{(16)}$になります。

しかし、いちいち、$1473_{(16)}$や$2405_{(16)}$という機械語を覚えることはできません。そこで、これらの機械語の意味を人間にとって分かりやすい言葉に置き換えます。この言葉を**ニーモニック**(mnemonic)と呼び、$14_{(16)}$に対応するニーモニックをLD、$24_{(16)}$に対応するニーモニックをADDA、$25_{(16)}$に対応するニーモニックをSUBAと言います。

**ポイント**

機械語を人間の分かる言葉に置き換えたものをニーモニックと呼ぶ。

さて、LDと言われても、最初のうちは、これ自体が意味不明の言葉であると感じられると思います。LDとは、**Load**(ロード)の略語で、「データを転送する」という意味合いで使うことが多く、この場合、汎用レジスタから汎用レジスタへのデータの転送(複写)という意味で使われます。なお、ADDAはADD Arithmetic(算術加算)の略で、SUBAはSUBtract Arithmetic(算術減算)の略です。

前述のように、$1473_{(16)}$という機械語は「レジスタ GR3 の内容をレジスタ GR7 に複写する」というものでした。14のあとの7と3は汎用レジスタの番号です。そこで、この機械語$1473_{(16)}$をニーモニックを使って、LD　GR7,GR3と表現します。慣れてくると、「LD　GR7,GR3」というプログラムを読んだだけで、「レジスタGR3の内容をGR7にロードするんだな」と理解できます。同様に、$2405_{(16)}$は「ADDA GR0,GR5」となります。

さて、これらの命令($1473_{(16)}$、$2405_{(16)}$)の語長(命令の長さ)は、16ビットですので、COMET IIの主記憶装置の1語の中に収まります。したがって、これらの命令の語長は1語となります。

それでは、次の命令を考えてみましょう。

主記憶装置8000番地の内容をレジスタGR4に格納せよ。

この命令の機械語は、3.2.1項に示したように10401F40$_{(16)}$となります。最初の10$_{(16)}$が主記憶装置の内容を汎用レジスタに転送する(つまり、ロード)命令を意味します。したがって、ニーモニックは、LDとなります(*1)。次の4がロード先の汎用レジスタの番号です。次の0は指標レジスタを使わないという指定ですが、これについては後述します。その次の1F40$_{(16)}$は、10進数では8000になり、これが転送されるデータが格納されたアドレスになります。

```
 LD        GR4                  アドレス
  ↓         ↓                      ↓
←──→                  ←──────────────────────→
1    0     4     0     1     F     4     0
                 ↑
            この0は指標レジスタを使わないという意味
```

この機械語を、ニーモニックを使って表現すると、LD　GR4,8000となります。あるいは、8000の部分を16進表記で、LD　GR4,#1F40と書くこともできます。この例のように、CASL IIでは16進数でデータを書くときには、#をその頭につけます。

さて、この命令(LD　GR4,8000)は、全部で32ビットの長さになっています。これを、COMET　IIの主記憶装置に格納するには2語分の領域が必要になります。最初の1040$_{(16)}$が1語目に格納され、次の1F40$_{(16)}$が2語目に格納されます。

**ポイント**

COMET IIには、32ビット(2語)の機械語命令と16ビット(1語)の機械語命令の2種類あります。

**演習問題3-2**

(1) 10501F00のニーモニックは何でしょうか。

(2) 「LD GR1,1000」を機械語に変換しなさい。

### 3.2.3　実際のCOMET IIの機械語

COMET IIの機械語のサマリを、情報処理技術者試験出題範囲から引用して以下に示します。

(*1) 14$_{(16)}$で始まる機械語もLDでした。このように複数の機械語を同じニーモニックで表現します。

表3-2　COMET IIの機械語の概要

| 種　類 | 説　明 | 対象となる機械語のニーモニック |
|---|---|---|
| ロード、ストア、ロードアドレス命令 | レジスタ間、レジスタ・記憶装置間での値の転送、アドレスのセット | LD, ST, LAD |
| 加算・減算演算命令 | データの加算・減算 | ADDA, ADDL, SUBA, SUBL |
| 論理演算命令 | ビット操作のための命令 | AND, OR, XOR |
| 比較演算命令 | データを比較してフラグレジスタをセット | CPA, CPL |
| シフト演算命令 | シフトに関する演算 | SLA, SRA, SLL, SRL |
| 分岐命令 | 条件によって制御の流れを変更 | JPL, JMI, JNZ, JZE, JOV, JUMP |
| スタック操作命令 | スタックの操作 | PUSH, POP |
| コール、リターン命令 | サブルーチンに関する命令 | CALL, RET |
| その他 | その他の命令 | SVC, NOP |

個々の命令の詳細に関しては、それぞれの説明の中で行います。また、第6章では、すべての機械語命令について詳細な説明を行っています。

## 3.3 コンピュータプログラムの動作

コンピュータのプログラムを動作させるためには、そのプログラムを主記憶装置に格納しなければなりません。プログラムは、通常は、ディスク装置などの補助記憶装置上に置かれ、実行するときに主記憶装置に転送されます。「転送」された後も、補助記憶装置上に元のプログラムはそのまま残っています。その意味では、「転送」というよりは「複写」といった方が適切です。プログラムを主記憶装置上に転送する作業もプログラムによって行われます。このプログラムを一般に、**ローダ**(loader)と呼び、この操作を**ロードする**といいます。なお、主記憶装置から汎用レジスタにデータを転送する場合や、汎用レジスタ間同士でデータを転送する場合も、ロードと呼んでいましたね。

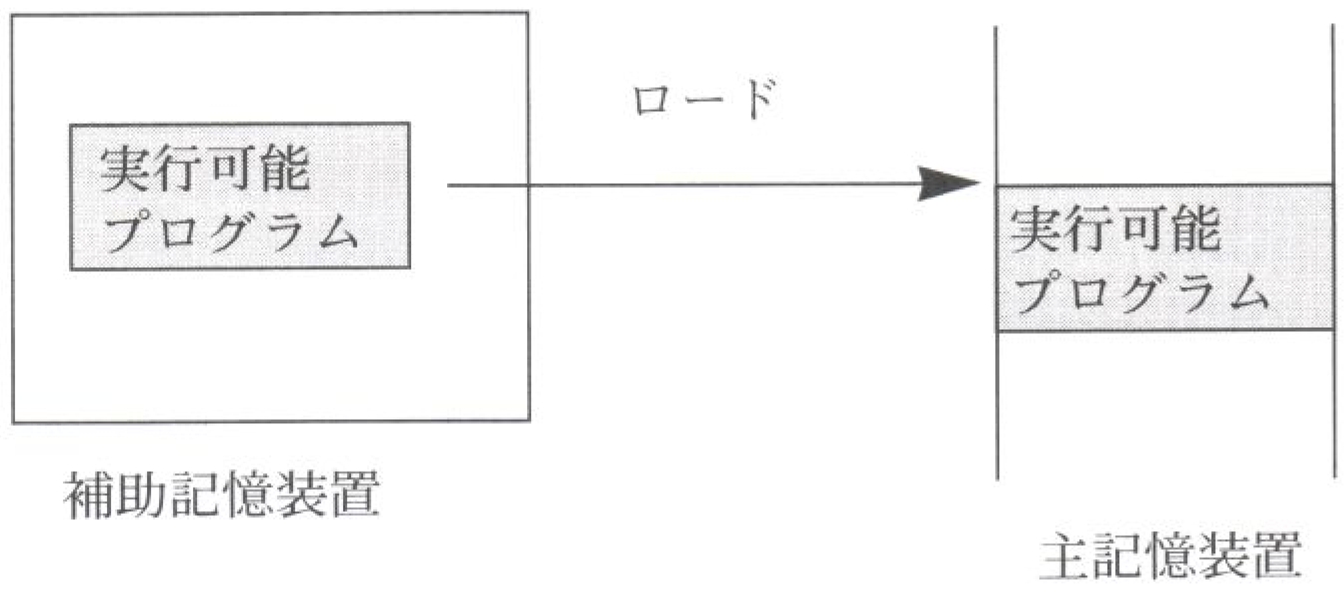



**図 3-5　ロードのイメージ**

パソコンの具体的な実例で考えてみましょう。Windows98でエクスプローラを起動したいとします。このプログラムは、通常はハードディスク(Windowsではドライブ番号で補助記憶装置を区別します。ここでは、一般的なCドライブであると仮定します)上に置かれます。より具体的に言うと、Cドライブの¥WINDOWSというフォルダの中にあるEXPLORER.EXEというファイルに格納されたプログラムが、エクスプローラの実行可能プログラムです。

スタートメニューからエクスプローラを選んだり、エクスプローラのアイコンをクリックすると、Windows98の機能によって、EXPLORER.EXEが主記憶装置に転送され、実行されます。

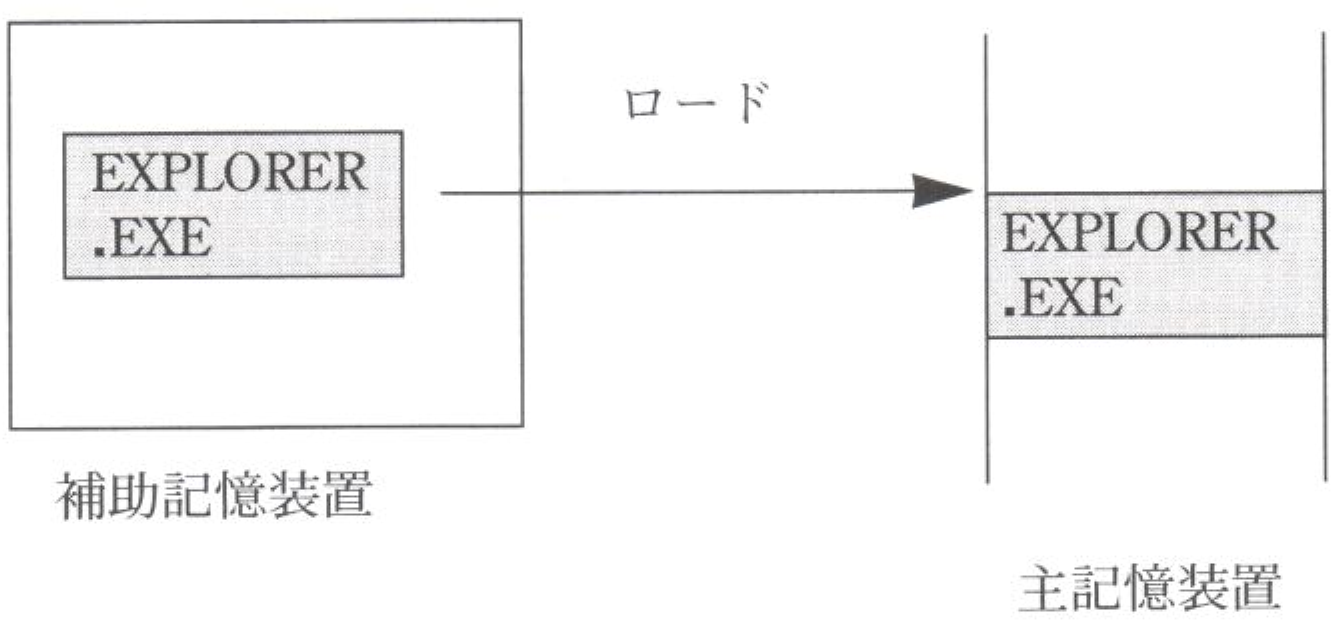



**図3-6　エクスプローラの実行のイメージ**

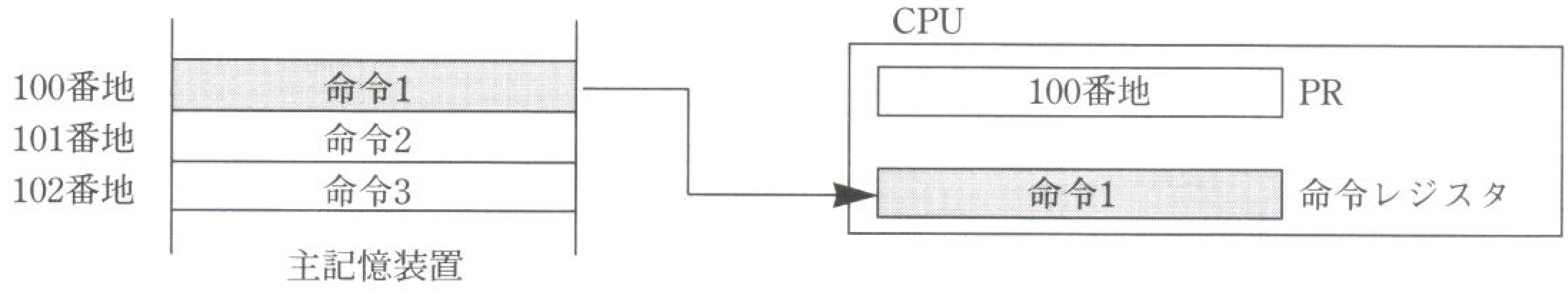
この状況は、COMET II も同じです。補助記憶装置上にある機械語のプログラムがローダによって主記憶装置に転送されます。主記憶装置上に転送されたプログラムは大きく分けて、実行を伴う機械語の部分と、プログラム中で定義されたデータが含まれています。

機械語の命令は、主記憶装置に格納されますが、前述のようにCOMET II の主記憶装置は、$2^{16}$=65536

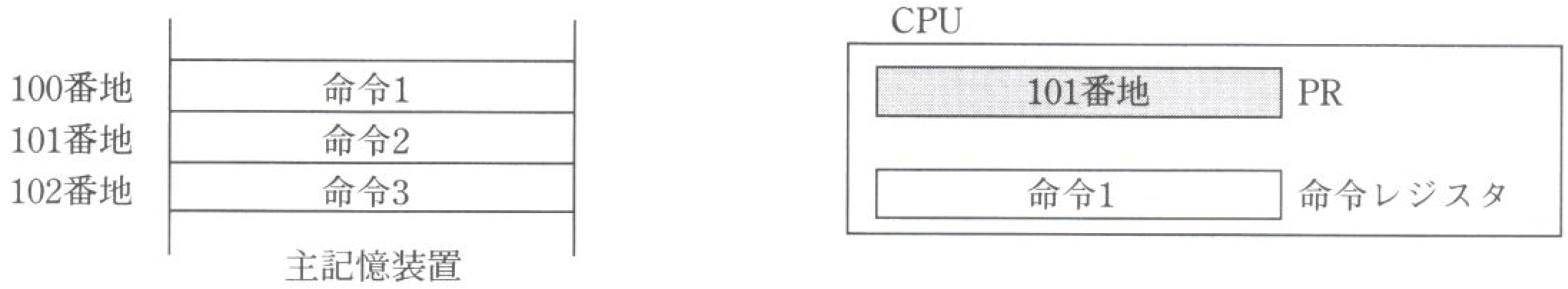
個の引き出しのついた入れ物であると考えることができます(図3-1参照)。この引き出しのことを語(word)と呼び、1語16ビットの容量であることは前述しました。各語には、先頭から順にアドレス(address)が付いていて、先頭が0番地、次が1番地、2番地と続き、最後が65535番地になることも学習しました。さらに、COMET IIの機械語には、16ビットと32ビットの2種類の語長の機械語があることもすでに学習しています。

さて、この機械語の命令はどのようにして実行されるのでしょうか。その仕組みはとても単純で、基本的にプログラムの先頭の機械語から次々に取り出されて実行されます。通常、1つの命令の実行が終了すると，その次の命令が取り出されて実行されます。つまり、図3-7に示すように、通常は、主記憶装置に格納された命令が、その順に取り出されて実行されているのです。

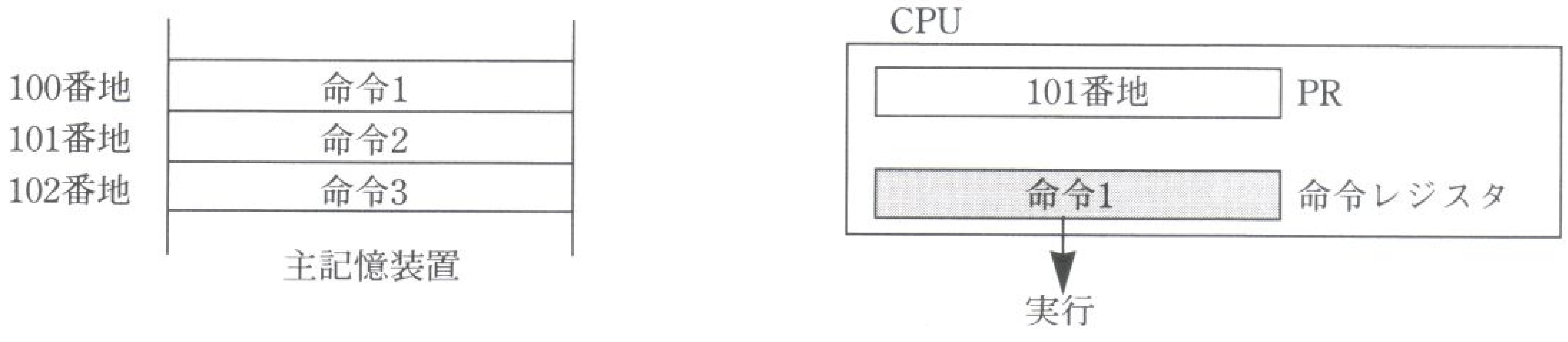
(1) PRが示している番地（100番地）の命令をCPUの中の命令を解読するレジスタにロードする。
COMET IIの仕様では、はっきりと記述されていませんが、このようなレジスタを一般に**命令レジスタ**と呼びます。




(2) PRの番地を次の命令のある番地(101番地)にする。
COMET IIには、1語で1つの命令を構成する機械語と2語で1つの命令を構成する機械語の2種類ありますが、説明の都合上、ここでは、1語で1つの命令を構成する機械語のみと仮定します。2語で1つの機械語を構成する場合、番地は＋1ではなく＋2されますので、102番地ということになります。




(3) 命令レジスタに格納した機械語を実行する




(4) 次の命令を取り出す：PRが示している番地の命令をCPUの中の命令を解読する命令レジスタにロードする。この手順は、(1)と同じです。したがって、これ以降の手順は、(2)→(3)→(1)→(2)→(3)→…と続いていきます。

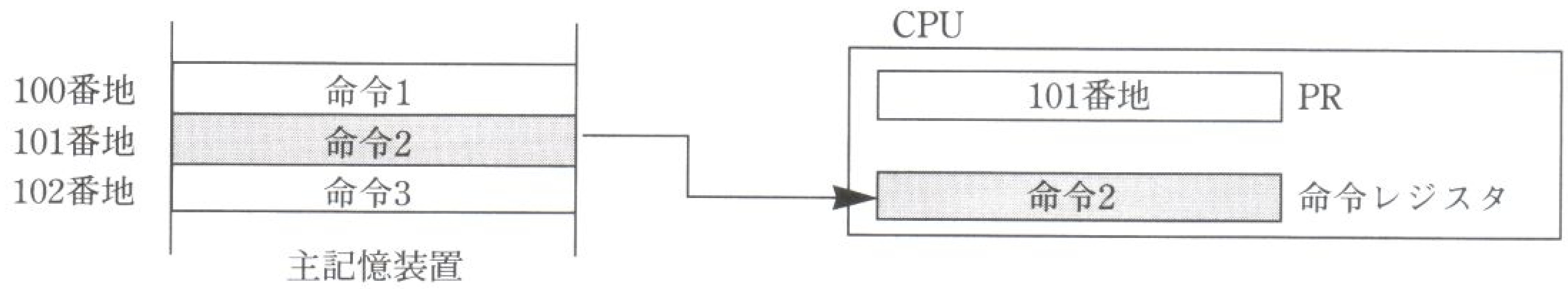



**図3-7　機械語の実行過程(直線的な処理の流れ)**

しかし、図3-7に示した方式では上から下へ1回しか各命令を実行できないので、本格的なプログラムを組むことができません。そのため、機械語の命令には、データを比較する命令とその比較結果に従って、別の機械語の命令へ処理をジャンプさせる**分岐命令**があります。この命令を組合わせることによって、複雑な条件判断処理や繰り返し処理を実行することが可能となります。分岐命令の詳細については、6.6節「分岐命令」で詳しく説明します。

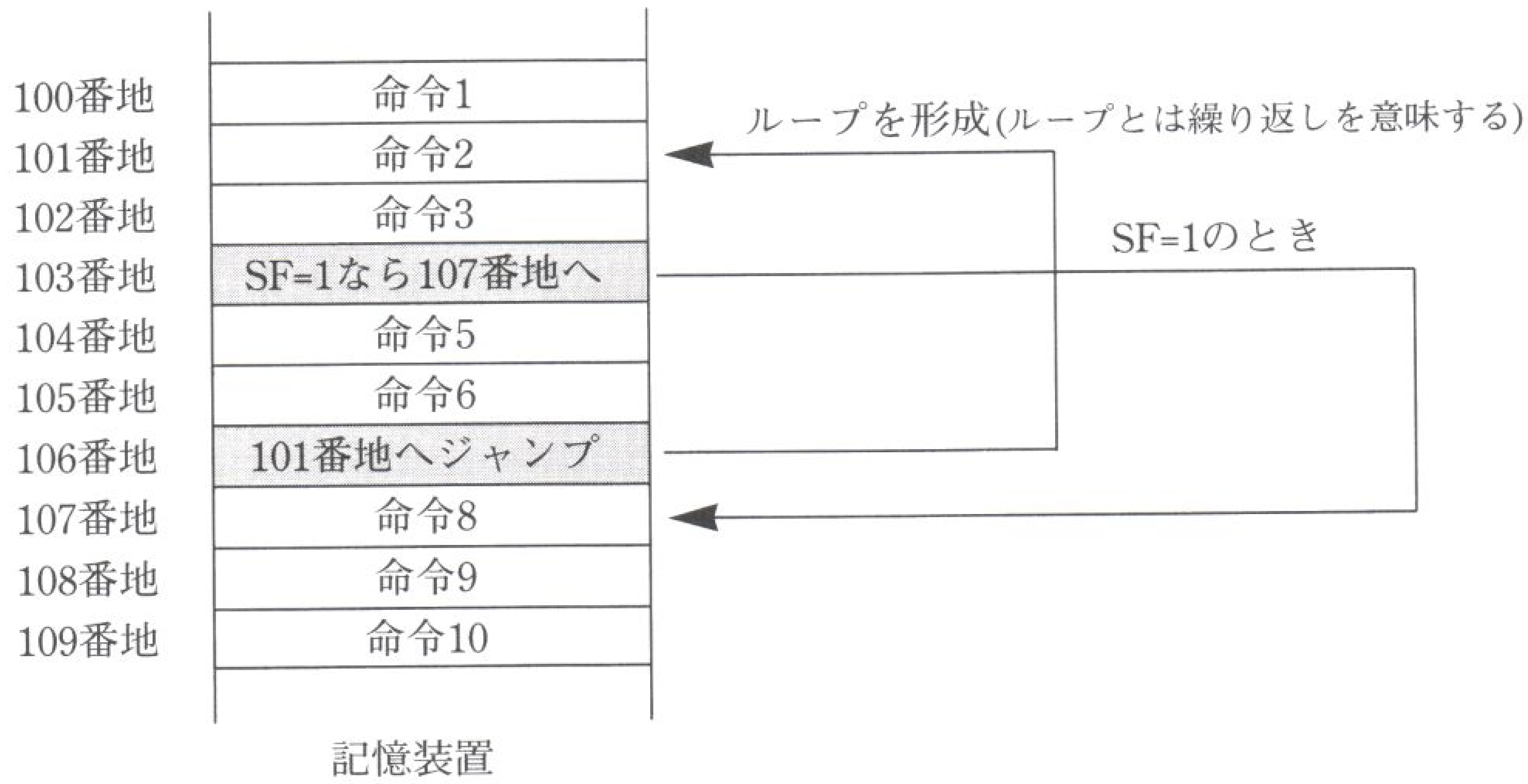



**図3-8 分岐命令による条件判断**

## ポイント

(1) COMET II では、主記憶装置に格納された機械語がPRに基づいて次々に取り出されて実行される。
(2) プログラムの流れを変えるには分岐命令を用いる。

## 演習問題3-3

(1) PRと実行している機械語の関係を説明してください。
(2) PRは16ビットです。ロードするプログラムが4000バイトのときに、COMET II でプログラムをロードできるアドレスの範囲を述べてください。
(3) COMET IIにおける機械語の実行過程を簡潔に説明してください。

## 3.4 アセンブラ言語とその役割

本章では、データの加算のみを行う基礎的なプログラムを用いて、具体的な機械語の概念とアセンブラ言語の基礎を示します。

### 3.4.1 LD, ST, ADDA, RET命令の概要

本節では、COMET II の命令の中でも、簡単なプログラムを作るのに最低限必要なLD命令、ST命令、ADDA命令を取り上げます。また、プログラムを終了させるための、RET命令についても言及します。

さて、3.1.3項で説明したように、**LD命令**は、主記憶装置上からデータを汎用レジスタへ転送する命令です。一方、**ST命令**は、LD命令とは逆に汎用レジスタの内容を主記憶装置上へ転送する命令です。なお、STはSTore(ストア)の略です。

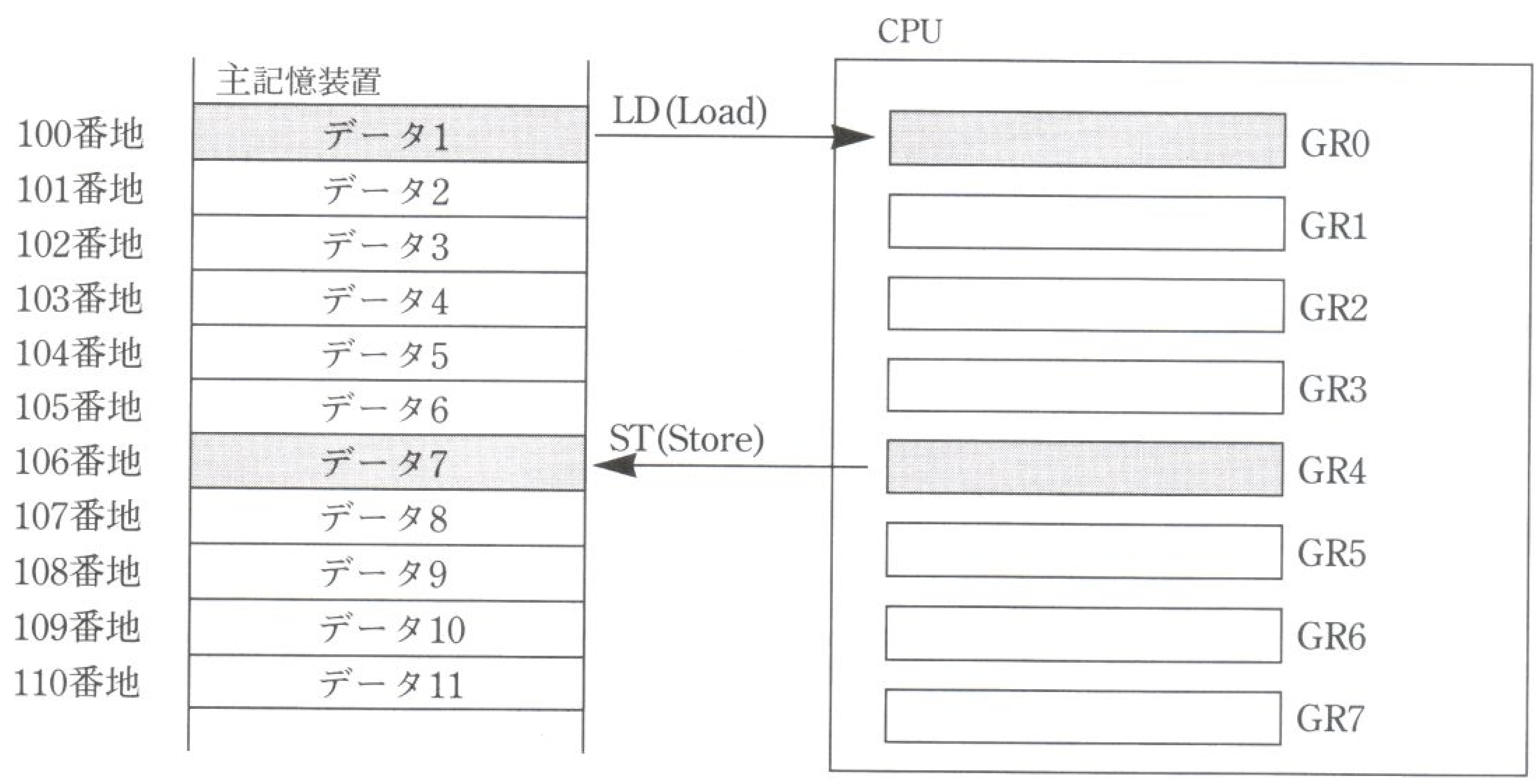



**図 3-9　LD命令とST命令**

**ADDA命令**は、汎用レジスタ上のデータと主記憶装置のデータ、あるいは汎用レジスタ間のデータを加えて、それを汎用レジスタに格納する命令です。

図3-10は、ADDA命令の動作を説明したものです。(1)は100番地の値とレジスタGR0の値を加えて、GR0に格納する様子を示し、(2)はレジスタGR2とGR4の値を加えて、GR2に格納する様子を示しています。この演算は、「GR4の値をGR2に加える」とみなすこともできます。

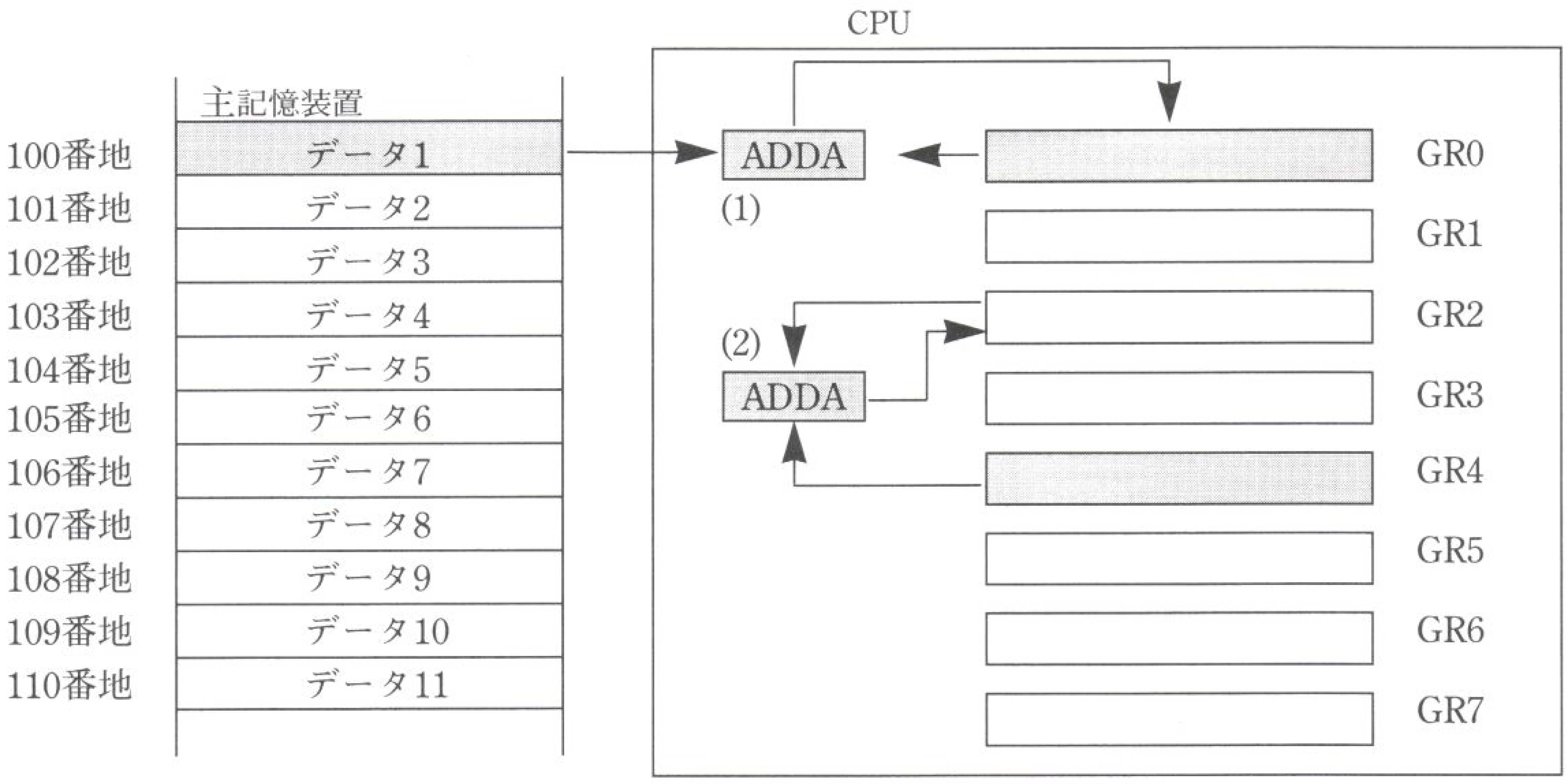



**図 3-10　ADDA命令**

**RET**は、サブルーチンの概念を知らないと正確な意味を説明できないのですが、ここでは、プログラムの実行を終了させる命令であると理解してください。

### ポイント

LDは、主記憶装置のデータを汎用レジスタにロード(転送)、あるいは汎用レジスタ同士のデータの転送に用いる。
STは、汎用レジスタの内容を主記憶装置にストア(格納)するのに用いる。
ADDAは、主記憶装置の値と汎用レジスタの値、あるいは汎用レジスタ同士の加算のために用いる。
RETは、プログラムを終了させるのに用いる。

## 3.4.2　機械語のプログラムによる説明

それでは、実際に、5+3=8を計算するプログラムについて考えてみましょう。ただし、主記憶装置の7番地には5が、8番地には3がすでに格納されているとします。9番地には何が格納されていてもプログラムの実行に影響を与えません。この例では、8029(16)という意味のない値が入っているとしますが、図は？マークで示しています。

| | |
|---|---|
| 7番地 | 5 |
| 8番地 | 3 |
| 9番地 | ? |

このプログラムの骨格は以下のようになります。

(1) レジスタGR1に主記憶装置の7番地に格納された値である5をロードする。

(2) レジスタGR1に主記憶装置の8番地に格納された値3を加える。
(3) レジスタGR1の値を主記憶装置の9番地に格納(ストア)する。
(4) プログラムを終了する。

さて、このプログラムの機械語を16進数で示すと次のようになります。しかしながら、この機械語を見て、何をやっているのか理解できる人間はまず世の中に存在しないのではないでしょうか。

```
10100007201000081110000981000005000038029
```

この機械語は、補助記憶装置(通常は、ハードディスク)中に置かれています。これを実行するためには、この機械語を補助記憶装置から主記憶装置にロード(転送)する必要があります。COMETの記憶装置は1語16ビットなので、これを主記憶装置の0番地から1語ずつ配置すると次のようになります(*1)。左の番号は、その語が配置されたアドレスを10進数で示しています。

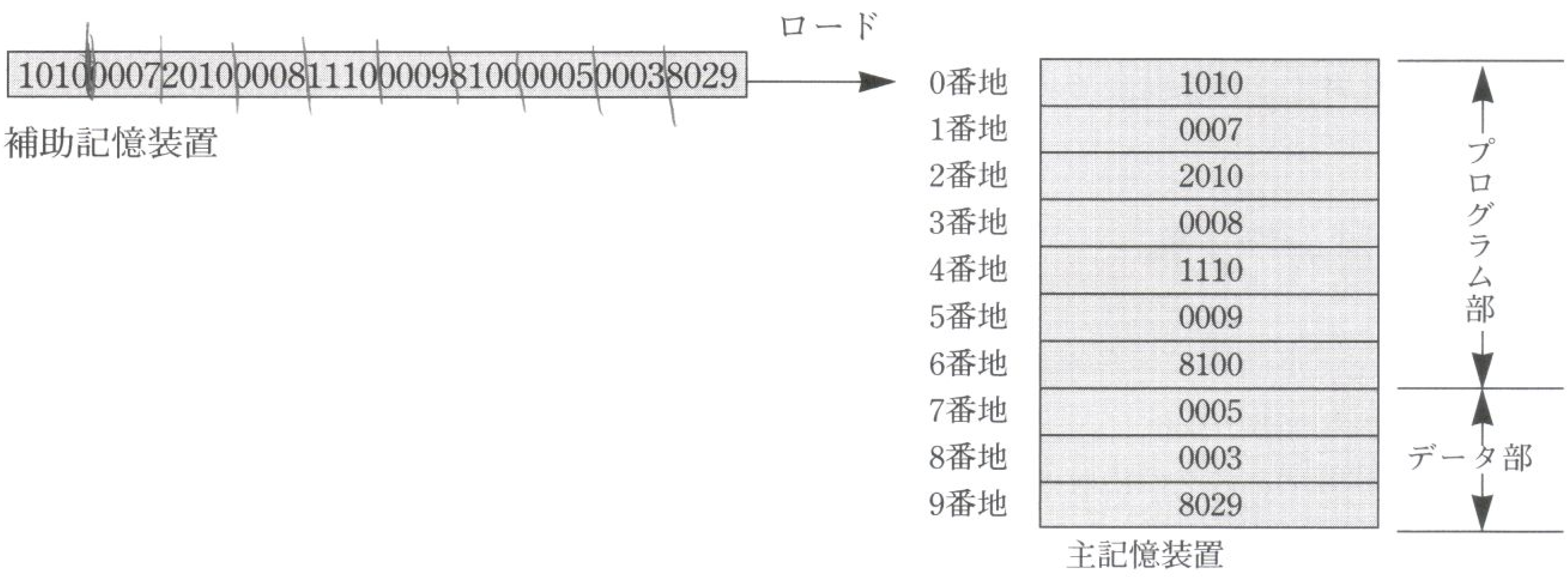



**図3-11 プログラムのロードと配置**

しかし、このように配置しても何を行っているのかよく分かりません。そこで、各機械語をまとめて、そこで何をやっているのかを横に示しましょう。このプログラムにおけるLD, ST, ADDAの機械語は、32ビットなので、2語必要になりますが、RETは16ビットなので1語必要になります。

(*1) 実際のコンピュータでは、私たちが作成したプログラムを0番地にロードして実行するということはまず考えられません。この領域はシステムの重要な目的に使用されることが多いからです。しかし、説明上、0番地からロードする方が単純なので、このようにしています。

| 番地 | 機械語 | 内容 | |
|---|---|---|---|
| 0： | 1010 | 7番地目に格納されたデータをレジスタの1番目(GR1)に | プログラム部 |
| 1： | 0007 | 転送(ロード)する。実行後、GR1の値は5となる。 | |
| 2： | 2010 | レジスタの1番目(GR1)の内容(5)と8番地目の内容(3)を加えて、 | |
| 3： | 0008 | GR1に格納する。実行後、GR1の値は8となる。 | |
| 4： | 1110 | レジスタの1番目(GR1)の内容(8)を9番地目に | |
| 5： | 0009 | 転送(ストア)する。9番地目の内容は8となる。 | |
| 6： | 8100 | プログラムの実行を終了する。 | |
| 7： | 0005 | 5の値が入っている部分。 | データ部 |
| 8： | 0003 | 3の値が入っている部分。 | |
| 9： | 8029 | 計算結果を格納する部分。 | |

このプログラムから分かるように、プログラムの機械語の部分とデータの部分が共に主記憶装置に格納されていることに注意してください。重要なことは、コンピュータから見た場合、主記憶装置に格納されたものがプログラムなのかデータなのかは区別がないということです。このプログラムをコンピュータが0番地から実行すれば正しく動作しますが、7番地から実行すれば0005(16)データをプログラムと勘違いして実行してしまいます。この場合は、プログラムの暴走(3.5節参照)という危険な状態になります。

**ポイント**

プログラムとデータは、共に主記憶装置上に区別無く格納される。

さて、このプログラムの実行前は、主記憶装置上の7～9番地の内容は、次のようになっていますね。

| 番地 | 値 |
|---|---|
| 7 | 0005 |
| 8 | 0003 |
| 9 | 8029 |

しかし、プログラムの実行後は、次のように9番地の値が8になります。

| 番地 | 値 |
|---|---|
| 7 | 0005 |
| 8 | 0003 |
| 9 | 0008 |

これで、9番地目に計算結果が格納されたことが分かるでしょう。

ところで、世の中には、時と場合によってどうしても機械語でプログラムを書かなければならない状況も存在します。CやJavaなどの**高水準言語**(*1)は、多くのコンピュータ上で動作するように汎用的に書かれているため、ある状況下では特定のコンピュータの能力を発揮することができないからです。しかし、いくら効率が良いから機械語でプログラムを作成する必要があるんだと言われても、この数値で書かれた訳の分からない機械語を見ていると寒気がする人も多いでしょう。この例では、高々、数個の機械語を扱っているだけですが、実際の開発では、数百、数千の機械語を扱うことになります。

そこで、**アセンブラ言語**というものが考案されました。これは、機械語を人間の理解できる言葉に1対1に対応させてプログラミングを可能にするものです。たとえば、機械語で$10_{(16)}$というのは、主記憶装置上の値を汎用レジスタに転送(ロード)するという意味ですが、これをロードの省略形LDで表現できるようにするということはすでに学習済みです。また、このLDをニーモニックと呼ぶことも学習しています。

さて、アセンブラ言語で、この機械語を表現すると次のようになります(*2)。

```
SAMP1   START
        LD      GR1,7
        ADDA    GR1,8
        ST      GR1,9
        RET
        DC      5
        DC      3
        DS      1
        END
```

もちろん、初めてこのプログラムを見る人にとっては、これでも何のことかさっぱりわからないでしょう。最初のSAMP1というのはこのプログラムに付けられた名前です。**START**というのは、「ここからアセンブラ言語の記述を開始しますよ」という意味で、機械語ではありません。したがって、対応する機械語は生成されません。

また、最後の**END**も「ここでプログラムの記述を終了しますよ」という意味で、対応する機械語は存在しません。したがって、これもまた対応する機械語は生成されません。このことは、最初のうちは分かり難い点ですが、今の段階では、STARTやENDは対応する機械語が無いのであるが、書かなくてはいけないものだと思っていてください。なお、このSTARTやENDのことをアセンブラ命令と呼びますが、これについては4.5節で述べます。

DCおよびDS命令はデータを定義する命令ですが後述します。

実際に、このプログラムをファイル(たとえば、SAMP1.caslというファイル名)に格納し、CASL IIアセンブラで機械語に変換して、COMET IIシミュレータで実行させることができます。Windows95/98/NT/2000をお使いの方はGUI版であるCASLDVがありますのでこの作業はボタン操作だけで行えます。その具体的な手順は付録に示されていますので、ぜひ、実際に動作を確認してみて

(*1)機械語およびそれをベースとして作成されているアセンブラ言語は、その構成要素が人間の考え方に比べて原始的なため、低水準言語と呼ばれます。これに対し、CやJavaなどの言語は、人間の考え方に近いため、高水準言語と呼ばれています。

(*2)アセンブラ言語の書き方は、第4章で正確に説明します。ここでは、プログラム中の1行目のSAMP1だけは、行の先頭(1カラム目)から記述し、それ以外は第2カラム目以降のどこからでも記述すればよいとだけ思ってください。

ください。

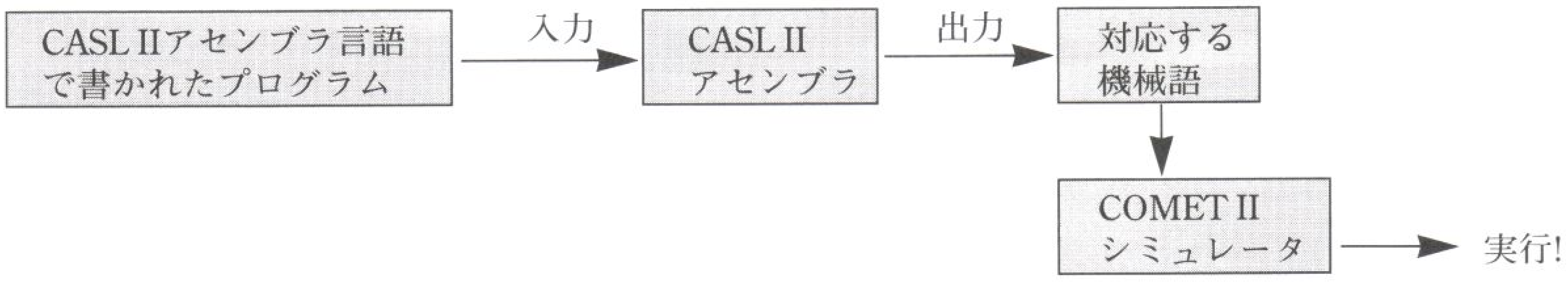



**図 3-12 アセンブラ処理の流れ**

参考のために、アセンブリ言語の横に対応する機械語を示します。

```
アセンブリ言語                       対応する機械語
                                     番地    機械語
SAMP1   START
        LD      GR1,7                 0      1010
                                      1      0007
        ADDA    GR1,8                 2      2010
                                      3      0008
        ST      GR1,9                 4      1110
                                      5      0009
        RET                           6      8100
        DC      5                     7      0005
        DC      3                     8      0003
        DS      1                     9      8029
        END
```

なお、SAMP1のプログラムSAMP1.caslは付属のCD-ROMの中に入っています。これをCASLDVにかけて実行したときの様子を以下に示します。

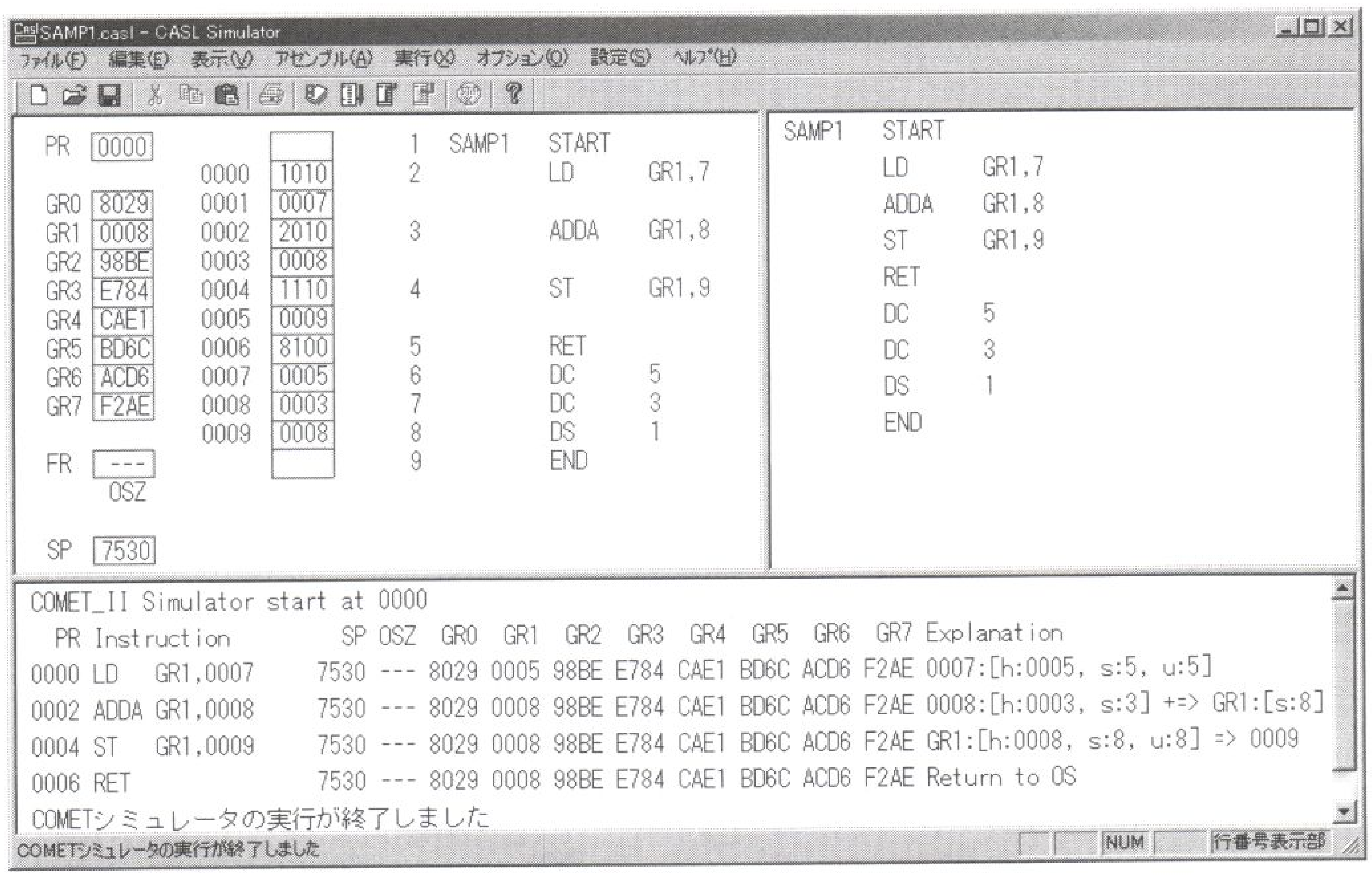



**図3-13 CASLDVの画面**

さて、「LD GR1,7」という意味は、何度も繰り返して説明しますが、主記憶装置上の7番地目の値をレジスタGR1に転送(ロード)せよという命令です。ロード命令を実行しても、主記憶装置の内容は、そのまま残っています。

「ADDA　GR1,8」という意味は、GR1の値(5)と主記憶装置上の8番地目の値(3)を加えて、その結果(8)を再びGR1に戻せという意味です。これでGR1の値は8となります。

「ST　GR1,9」という命令は、GR1の値(8)を主記憶装置上の9番地目に格納せよという意味です。

「RET」命令は、プログラムの実行を終了させます。

「DC　5」という命令は、このアドレス(RET機械語命令が配置された後のアドレスのことで、7番地になっている)の内容に(機械語の命令を置くのではなくて)、データとして5を配置せよという意味です。つまり、7番地目の内容が値5になります。

「DS　1」は、「そのアドレス(この場合、9番地)から1語分確保せよ」という命令ですが、これについては後述します。

## 演習問題3-4-2

プログラムSAMP1を参考にして、7番地に格納されたデータを2倍して8番地に格納するプログラムを書いてください。

### 3.4.3　ラベルの利用

ところで、このプログラムを多少変更することを考えます。計算した結果を9番地だけでなく、最初のデータがあった部分(7番地)にも入れることにしましょう。プログラムを単純に手直しすると、次のようになります。

```
SAMP2   START
        LD      GR1,7
        ADDA    GR1,8
        ST      GR1,9
        ST      GR1,7   ←――― : 追加部分
        RET
        DC      5
        DC      3
        DS      1
        END
```

しかし、これではうまくいきません。機械語のプログラムを横に示してみましょう。

```
アセンブリ言語                      対応する機械語
                                    番地    機械語
SAMP2   START
        LD      GR1,7               0       1010   ┐
                                    1       0007   │
        ADDA    GR1,8               2       2010   │
                                    3       0008   │ プログラム部
        ST      GR1,9               4       1110   │
                                    5       0009   │
        ST      GR1,7               6       1110   │
                                    7       0007   │
        RET                         8       8100   ┘
        DC      5                   9       0005   ┐
        DC      3                   10      0003   │ データ部
        DS      1                   11      8029   ┘
        END
```

上記の機械語の番地(アドレス、10進表記)を見ると、一目瞭然なのですが、1つの機械語命令を入れてしまったために、データを配置しているアドレスがずれてしまったのです。この例では、データは9番地(上のプログラムのDC　5の部分)から格納されていることになります。

このプログラムを強引に実行すると、次のように−32505というとんでもない数値が計算されます。

| | | |
|---|---|---|
| LD | GR1,7 | 7番地目の値($0007_{(16)}$)をGR1にロードする。GR1の値は$0007_{(16)}$になる。 |
| ADDA | GR1,8 | 8番地目の値($8100_{(16)}$＝−32512)をGR1に加える。GR1の値は、$8107_{(16)}$＝−32505になる。 |
| ST | GR1,9 | 9番地目にGR1の値をストアする。9番地目の値は$8107_{(16)}$＝−32505になる。 |
| ST | GR1,7 | 7番地目にGR1の値をストアする。7番地目の値は$8107_{(16)}$＝−32505になる。 |
| RET | | プログラムを終了する。 |

つまり、7番地に入っている値は機械語「ST GR1,7」の第2語目($0007_{(16)}$)であり、この値がGR1にロードされます。8番地にはRET命令の機械語($8100_{(16)}$, 符号付きの10進数では−32512)が入っています。このRET命令の機械語($8100_{(16)}$＝−32512)を間違えてデータとみなして計算してしまったため、とんでもない値(−32505)が格納されてしまうのです。

そこで、つぎのように対象となるアドレスを修正する必要があります。このプログラムは正しく動作します。

```
SAMP3   START
        LD      GR1,9   ←──── 修正部分
        ADDA    GR1,10  ←──── 修正部分
        ST      GR1,11  ←──── 修正部分
        ST      GR1,9   ←──── 修正部分
        RET
        DC      5
        DC      3
        DS      1
        END
```

プログラムの実行前と実行後では主記憶装置内の内容は次のように変化します。

```
  実行前 ────────► 実行後
 番地  内容         番地  内容
  9    0005          9    0008
 10    0003         10    0003
 11    8029         11    0008
```

プログラムのアドレスは、この例のようにプログラムを修正するたびにずれます。プログラムの修正は、プログラム開発時にはつきものですから、そのたびごとにアドレスの値を修正していたのでは大変であり、場合によってはミスも発生します。

そこで、次のように書くことができます。

```
SAMP4   START
        LD      GR1,A
        ADDA    GR1,B
        ST      GR1,C
        RET
A       DC      5
B       DC      3
C       DS      1
        END
```

このプログラムでは、直接アドレスを書かずに、A, B, Cという名前を使っています。これを**ラベル**(label)と呼びます。ラベルは、アセンブラによって、翻訳時に対応するアドレスに変換されます。

この処理内容について説明します。アセンブラはプログラムを先頭の行から1行ずつ入力して機械語に変換していきます。ところが、アセンブラが、最初に「LD　GR1,A」を機械語に変換しようとしても、この時点ではAのアドレスがわかりません。そこでこの処理を保留して次に進みます。しかし、アセンブラは「LD　GR1,A」、「ADDA　GR1,B」、「ST　GR1,C」という各命令が2語必要で「RET」が1語必要であることはわかりますから、この部分の機械語命令に7語必要であることがわかります。「A　DC　5」の部分でAというラベルのアドレスが決定されます。この命令の前には7語あるのですから「A　DC　5」は8番地に配置されることになります。したがって、ラベルAには8番地というアドレスが割り当てられます。また、同様に、ラベルBには9番地、ラベルCには10番地というアドレスが割り当てられます。そのため、保留していた「LD GR1, A」の部分を、正しく「LD GR1,8」と変換することができるのです。

このプログラムをアセンブラにかけても、出力される機械語はSAMP1とまったく同じです。

また、このプログラムをSAMP3と同じように修正する場合、アセンブラがラベルのアドレスを計算し直してくれるので、追加部分だけを修正すればよいのです。SAMP5のプログラムでは、ラベルAには9番地、ラベルBには10番地、ラベルCには11番地というアドレスが割り当てられます。

```
SAMP5   START
        LD      GR1,A
        ADDA    GR1,B
        ST      GR1,C
        ST      GR1,A   ◀──── : ここだけ追加修正
        RET
A       DC      5
B       DC      3
C       DS      1
        END
```

本書に付属のCASLDV(CASL IIアセンブラとCOMET IIシミュレータ)を使って、ここに書いてあることをぜひ、確認してください。

### 3.4.4 ラベルの文法上の書き方とその意味

ラベルは、1文字から8文字までの英大文字(A〜Z)と数字(0〜9)を組み合わせた名前で、最初の文字は英大文字でなければなりません。また、GR0, GR1, GR2, GR3, GR4, GR5, GR6, GR7は、汎用レジスタを表すので、ラベルとしては使用することができません。

以下に、ラベルとして正しい名前と正しくない名前を示します。

**表3-3 正しいラベルの例**

| 正しいラベル | A ABC XYZ LONGNAME X1 X1Y1 |
|---|---|

**表 3-4 正しくないラベル**

| 正しくないラベル | 理 由 |
|---|---|
| A-1 | ハイフン(-)がある |
| 1ABC | 数字で始まっている |
| X_1 | アンダーライン(_)がある |
| TOOLONGNAME | 8文字を超えた |
| Aa | 英小文字がある |
| カナ | 漢字や片仮名は使えない |
| A@X | 英数字以外の文字がある |
| GR1 | GR1は、予約されていてラベルの名前としては使えない |

ラベルを定義するときは、プログラムの第1カラム目からラベルを書きます。第2カラム以降から書くと、それは、ラベルの引用（参照）かあるいはラベル以外の命令語とみなされますので、注意してください。

以下のプログラムでは、SAMP6, ADDSTEP, VALUEX, VALUEY, VALUEZがすべて1カラムから記述されており、これらがラベルの定義となります。SAMP6は、STARTアセンブラ命令に記述されているので、このプログラムの名前になります。ADDSTEPは、ADDA命令についており、この命令が格納されたアドレスを意味します。このように実行する命令にもラベルは付けられます。また、VALUEX, VALUEY, VALUEZは、データ部分に付けられた名前です。

```
SAMP6    START
         LD       GR1,VALUEX
ADDSTEP  ADDA     GR1,VALUEY
         ST       GR1,VALUEZ
         RET
VALUEX   DC       5
VALUEY   DC       3
VALUEZ   DS       1
         END
```

ラベルの定義（SAMP6、ADDSTEP、VALUEX、VALUEY、VALUEZ）

ラベルの引用（VALUEX、VALUEY、VALUEZ）

↑
第１カラムめ

なお、ラベルには、次のような名前も付けられますが、紛らわしいので、他の命令語と同じ名前は付けないほうがよいでしょう。

```
SAMP7   START
        LD      GR1,ADDA
        ADDA    GR1,DC
        ST      GR1,DS
        RET
ADDA    DC      5
DC      DC      3
DS      DS      1
        END
```

## 演習問題3-4-4

(1) ラベルの必要性について説明してください

(2) 次のラベルには誤りが含まれています。それを指摘してください

(a) 10DATA

(b) X_1

(c) Z-1

(d) $\alpha$

(e) SAMPLEPGM

### 3.4.5　オペレータとオペランド

機械語命令は、次のような構成をしています。

LD　　GR1,A

このとき、LDのことを**オペレータ**(operator)、GR1,Aのことを**オペランド**(operand)と呼びます。

```
オペレータ    オペランド
   ↓             ↓
   LD          GR1,A
```

オペレータとは操作を行う内容を示すもので、オペランドとは操作対象となるものを示します。

なお、見やすくなるからといって、CASL IIではオペランドの中に空白を入れてはいけません。たとえば、「GR1, A」とすると、CASLアセンブラはエラーメッセージを出力します。ただし、オペレータとオペランドの間にはいくつ空白を入れてもかまいません。

## 3.5 オペレーティングシステムの支援

さて、本章では、簡単な機械語の実行の様子を見てきました。しかし、この機械語のプログラムだけでは、コンピュータの中で動作することはできません。多くの場合、オペレーティングシステムの支援が必要です。たとえば、機械語命令を補助記憶装置の中から主記憶装置へローディングする処理も、実は、オペレーティングシステムが行います。

COMET IIにおいて、機械語プログラムが動作するためには、次の手順が必要になります。

| | |
|---|---|
| (1) 入出力のために、入出力装置を割り当てる | ↑ |
| (2) 機械語プログラムを記憶装置へローディングする | オペレーティングシステム |
| (3) 機械語プログラムを呼び出す | ↓ |
| | ↑ |
| 機械語プログラムを実行する<br>RETでオペレーティングシステムへ戻る | 機械語プログラム |
| | ↓ |
| | ↑ |
| (4) 入出力装置の割り当てを解除する | オペレーティングシステム |
| | ↓ |

本書に付属のCASLDV(COMET II シミュレータ)を起動する場合には、そのシミュレータがオペレーティングシステムということになるでしょう。

---

*FAQ (Frequently Asked Questions)*

プログラムでは、命令の部分とデータの部分の区別がなく、主記憶装置にロードされて実行されると学習しました。それでは、もし、次のように間違えてRET命令を除いたプログラムを実行するとどうなるのでしょうか。

```
SAMP8   START
        LD      GR1,A
        ADDA    GR1,B
        ST      GR1,C
A       DC      5
B       DC      3
C       DS      1
        END
```

*Answer*

ST命令までは、問題なく動作します。問題は、ST命令終了後に起きます。

COMET IIは、ST命令の次の命令を実行しようとして、データであるA番地の内容を機械語であると勘違いし、命令レジスタにロードします。そこには、5が格納されています。COMET IIは、これを

解読して実行しようとします。ところが、5に相当する機械語命令はありません。この場合、何らかのエラーが起きたことがわかります。

しかし、Aの部分にたまたま、機械語として成立する数値が入っていると、コンピュータはその数値を機械語として実行してしまいます。しかし、それは、もともと意図した正しいプログラムではありません。何か変なことをやらかして、そのまま、次の命令を実行します。

このように、意図しない部分をプログラムとみなして、次々と無意味な処理を実行してしまいます。この状況を、**プログラムの暴走**と呼んでいます。プログラムの暴走によって、意図しないことが起こり、事故が起こることもあります。ただし、本書に付属のCASLDVは、標準の設定では、このエラーを検出するようになっています。

## 総合問題3

**[演習3-1]** CASL II アセンブラとCOMET IIシミュレータの役割について解説してください。

**[演習3-2]** 3.4.2節の最後の演習問題で作ったプログラムを実際に、CASLDVを用いて、CASL II アセンブラにかけてCOMET IIシミュレータで実行してください。

**[演習3-3]** 次の文章の正誤を判定してください。

(1) すべてのレジスタは16ビットである。
(2) 全ての汎用レジスタは区別なく全く同じに使える。
(3) プログラムを補助記憶装置から主記憶装置にロードすると補助記憶装置上のプログラムは消えてしまう。
(4) COMET IIの機械語は全て2語である。
(5) ST命令は汎用レジスタの内容を主記憶装置に転送する。
(6) プログラム部とデータ部は主記憶装置上では区別がないのでデータ部をプログラムとまちがえて実行しないようにすべきである。
(7) オペランドの中に空白を入れてはいけない。
(8) A_1はラベルとして正しい名前である。

**[演習3-4]** 次の文章の空欄にあてはまる語句を入れてください。

(1) COMET IIの主記憶装置は [ (a) ] 番地から [ (b) ] 番地の語がある。
(2) 汎用レジスタの個数は [ (c) ] 個である。
(3) FRは [ (d) ] ビットである。
(4) ラベルの長さは [ (e) ] 文字までが許される。
(5) COMET IIのプログラムが動作するためには [ (f) ] の支援が必要である。

# *Memorandum*

# 第4章 アセンブラ言語の書き方の基礎

## 4.1 アセンブラ言語の書き方の基本

COMET IIは、前章で説明したように、CASL IIというアセンブラ言語でプログラムを作成することを前提としています。本章では、このアセンブラ言語の書き方について説明します。例を用いて説明しましょう。

次のプログラムは、前章で学習したプログラムです。

```
第一カラム目
↓
SAMP4    START
         LD        GR1,A
         ADDA      GR1,B
         ST        GR1,C
         RET
A        DC        5
B        DC        3
C        DS        1
↑        END
第一カラム目
```

まず、ラベルを定義するときは必ず1カラム目から書きます。第3章で学習したように、ラベルは英大文字で始まり、その後に英大文字か数字が続きますが、全部で8文字以下でなければなりません。また、STARTからENDまでの1つのプログラムの中で同じ名前のラベルを重複して定義してはいけません。次の例は誤りです。

```
X  DC  5
X  DC  6
```

機械語命令の動作を示す部分のことをオペレータと呼び、ニーモニックで示します(これも第3章で学習済み)が、この部分は必ず2カラム目以降に書きます。また、オペレータの後に記述された部分(たとえば、LDの「GR1,A」)をオペランドということも学習しました。オペレータとオペランドの間は1つ以上の空白で区切ります。しかし、オペランドの内部には空白を入れてはいけません。たとえば、「GR1,　A」としてはいけません。

さて、次ページの左右のどちらのプログラムも、上記の規則を忠実に守っています。

```
SAMP4  START
  LD   GR1,A
       ADDA   GR1,B
  ST GR1,C
              RET
A DC 5
B   DC        3
C      DS     1
 END
```

```
SAMP4   START
        LD      GR1,A
        ADDA    GR1,B
        ST      GR1,C
        RET
A       DC      5
B       DC      3
C       DS      1
        END
```

でも、左のプログラムは、とても読み難いと感じませんか。そこで、通常は、右のプログラムのようにラベルを第1カラム目から書き、その後は適当な個数の空白を空けて、さらにオペレータとオペランドの記述開始位置を合わせるという書き方がよく行われます。

それでは、表4-1にこのプログラムと、アセンブラによって生成された機械語を示します。

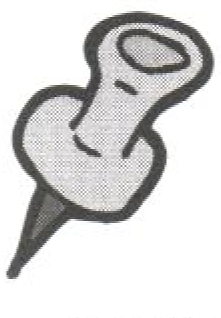
**表 4-1 アセンブラ言語と対応する機械語**

| アセンブラ言語 | | | 対応する機械語 | | | | |
|---|---|---|---|---|---|---|---|
| | | | 番地 | 機械語 | 番地 | 機械語 | 命令の語数 |
| SAMP4 | START | | | | | | |
| | LD | GR1,A | 0 | 1010 | 1 | 0007 | 2語 |
| | ADDA | GR1,B | 2 | 2010 | 3 | 0008 | 2語 |
| | ST | GR1,C | 4 | 1110 | 5 | 0009 | 2語 |
| | RET | | 6 | 8100 | | | 1語 |
| A | DC | 5 | 7 | 0005 | データ | | |
| B | DC | 3 | 8 | 0003 | データ | | |
| C | DS | 1 | 9 | 0000 | データ | | |
| | END | | | | | | |

この例から分かるように、1行に1つの命令を記述します。2行に渡って1つの命令を記述することはできません。たとえば、「LD　GR1,A」というのは「10100007(16)」という機械語の命令です。この機械語は、32ビットなので2語に渡って記憶装置内に置かれますが、1つの機械語命令は、プログラム上では必ず1行で書かなければなりません。

つまり、次のように書いてはいけません。

```
LD
           GR1, A        ←このようには書けない
```



**ポイント**

CASL IIでは1つの機械語命令は必ず1行で書かなければなりません。

なお、プログラム中にもあるように、プログラムには、START, END, DS, DCの命令を書きますが、これらを**アセンブラ命令**と呼びます。また、入出力のためにIN, OUTの命令が用意されていますが、これらを**マクロ命令**（詳細は後述）と呼びます。これ以外の命令を**機械語命令**と呼びます。

**表4-2　命令の種類**

| 命令の種類 | ラベル | 形　式 | | 説明個所 | 意　味 |
|---|---|---|---|---|---|
| アセンブラ命令 | ○ | START | [実行開始番地] | 4.5.1 | プログラムの先頭を定義 |
| | × | END | | 4.5.2 | プログラムの終わりを明示 |
| | △ | DS | 語数 | 4.5.4 | 領域を確保 |
| | △ | DC | 定数 [,定数] | 4.5.3 | 定数を定義 |
| マクロ命令 | △ | IN | 入力領域,入力文字長領域 | 4.6.1 | 入力装置から文字データを入力 |
| | △ | OUT | 出力領域,出力文字長領域 | 4.6.2 | 出力装置へ文字データを出力 |
| 機械語命令 | △ | 第6章を参照 | | | |

ラベルの意味：○:必ずラベルを指定しなければならない、△：指定してもしなくてもよい、×：指定してはならない

## 演習問題4-1

アセンブラの書き方に対する注意事項を指摘してください。

## 4.2 数字の書き方

プログラム中に、数字を書くことが必要な場合があります。たとえば、「A DC 5」という場合や「LD GR1,8」(GR1に8番地の内容をロードする)という命令の5や8がそれに相当します。このような数字は、固定された値を示すことから**定数**(constant)と呼ばれます。

CASL II では、数の定数には、10進法で示される**10進定数**、16進法で示される**16進定数**の2つがあります（この他に、文字定数やアドレス定数があります。表4-7参照）。

**表 4-3 数の定数**

| 定数の種類 | 記述法 | 説　　明 |
|---|---|---|
| 10進定数 | n | nは、−32768〜32767の数値とする。もし、nがこの範囲外の場合、その下位16ビットが有効となる。 |
| 16進定数 | #h | hは、4桁の16進数(16進数であるから、0〜9、A〜Fの数字を使う)である。したがって、必ず #0000〜#FFFFの間になる。 |

10進定数は、−32768〜32767の範囲しか書けないことに注意してください(*1)。もし、80000と書いた場合、その下位16ビットが取られますので、14464として扱われます(*2)。

いくつか例を示します。

```
            LD      GR1,2000     2000番地の内容をGR1に格納
            LD      GR1,#0700    700(16)番地(=1792番地)の内容をGR1に格納
            LD      GR1,#FF0A    FF0A(16)番地(=65290番地)の内容をGR1に格納
A           DC      −2000        Aのアドレスに値−2000を初期値として置く
B           DC      #7FFF        Bのアドレスに値32767を初期値として置く
```

16進定数のhは必ず4桁で記述します。#700と書くと誤りになります。

### 演習問題4-2

(1) 10進定数に100000と記述すると、実際には、どんな数が設定されますか。

(2) 30000を16進定数で記述してください。

(3) −1を16進定数で記述してください。

(*1) ただし、アドレスを表現する場合（表6-2のadr)、0〜65535の値を指定できます。なお、本書に付属のCASLDVでは、32768〜65535の間の数値は符号無しデータとして扱い、正しい2進数に変換されます。

(*2) 第2章の総合問題の[演習2-5]参照

## 4.3 コメントの書き方

プログラムに対する説明を**コメント**(comment)と呼びます。次の例は、SAMP1にコメントを入れたプログラムSAMPCです。

```
;コメント付きのプログラム
;                   作成者：内田智史、Sept. 28, 2000
;
;C ← A ＋ Bを計算するプログラム
;
SAMPC   START
        LD      GR1,A   GR1 ← A
        ADDA    GR1,B   GR1 ← GR1＋B
        ST      GR1,C   GR1 → C
        RET              ; OSに戻る
A       DC      5       Aの値
B       DC      3       Bの値
C       DS      1       Cの領域を1語分確保
        END
```

最初の5行は、行全体がコメントになっています。これを**注釈行**と呼びます。注釈行のときには、最初にセミコロンを書きます。セミコロンの前に空白があっても構いません。その後に、コメントを書きます。コメントは、アセンブラ言語の処理には、まったく影響を与えないので、何を書いてもOKです。また、コメントには上記に示すように漢字を書くことができます。

アセンブラ言語の記述にコメントを書く場合には、その命令にオペランドが記述されているかどうかで書き方が異なります。その規則を表4-4に示します。

**表 4-4　コメントの書き方**

| 行の種類 | | 記 述 の 形 式 |
|---|---|---|
| 注 釈 行 | | [空白]{;}[コメント] |
| 命令行 | オペランドあり | [ラベル]{空白}{命令コード}{空白}{オペランド} [{空白}[コメント]] |
| | オペランドなし | [ラベル]{空白}{命令コード} [{空白}[{;}[コメント]]] |

[ ] はその中の指定が省略できることを示し、{ }はその中の指定が必須であることを示しています。

表4-4によると、オペランドがない場合(RET命令やEND命令)には、注釈行と同じように、セミコロンが必要ですが、オペランドのある命令(LD命令やADDA命令)には、セミコロンが不要になり、命令記述後、直ちにコメントを書くことができます。以下の例で網掛けの部分がコメントです。

```
LD      GR1,A   GR1 ← A
ADDA    GR1,B   GR1 ← GR1＋B
ST      GR1,C   GR1 → C
```

もちろん、オペランドがあっても、コメントにセミコロンを書いても構いません。その場合、セミコロンがコメントの一部として認識されるだけです。

```
LD      GR1,A    ; GR1 ← A
ADDA    GR1,B    ; GR1 ← GR1＋B
ST      GR1,C    ; GR1 → C
```

ただし、RET命令にコメントをつけるときには、セミコロンをはずすとエラーになりますので注意してください。

```
RET     OS(オペレーティングシステム)に戻る ← セミコロンがないのでエラー
RET     ;OS(オペレーティングシステム)に戻る ← 正しい
```

これは、RET命令がオペランドを取らないため、「オペランドなしの場合にはコメント；を書いてからを書く」という規則（表4-4）に従うからです。

## 演習問題4-3

コメントの使い方で誤りのあるプログラムの番号とその理由を述べてください。
または、エラーにはならないが紛らわしい記述があれば指摘してください。

```
(1) LD     GR1, TEMP    セットするデータはTEMP
(2) LD     GR1, X       ,GR3
(3) RET    SUBROUTINE
(4) START  ;ここからスタート
```

## 4.4 文字コード

CASL IIで、使える文字コードは、JIS X 0201ローマ字・片仮名用8単位符号で規定する文字の組を使います。表4-5にその文字コードの一覧を示します。これ以外の文字が試験などで必要になった場合には、別途指示をすることになっています。

**表 4-5 文字コード表**

| 行＼列 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|
| 0 | 間隔 | 0 | @ | P | ` | p |
| 1 | ! | 1 | A | Q | a | q |
| 2 | " | 2 | B | R | b | r |
| 3 | # | 3 | C | S | c | s |
| 4 | $ | 4 | D | T | d | t |
| 5 | % | 5 | E | U | e | u |
| 6 | & | 6 | F | V | f | v |
| 7 | ' | 7 | G | W | g | w |
| 8 | ( | 8 | H | X | h | x |
| 9 | ) | 9 | I | Y | i | y |
| 10 | * | : | J | Z | j | z |
| 11 | + | ; | K | [ | k | { |
| 12 | , | < | L | ¥ | l | \| |
| 13 | - | = | M | ] | m | } |
| 14 | . | > | N | ^ | n | ~ |
| 15 | / | ? | O | _ | o | |

表4-5において、文字コードを16進数で表現したとき、行がその1桁目、列が2桁目を意味します。たとえば、「間隔」文字は空白のことを意味しますが、この文字コードは行が0、列が02なので、$20_{(16)}$＝32となります。また、数字の「5」は$35_{(16)}$＝53となり、英字の「A」は$41_{(16)}$＝65、「a」は$61_{(16)}$＝97となります。

## 4.5 アセンブラ命令

CASL IIでは、START, END, DC, DSという各命令を**アセンブラ命令**と呼びます。

表 4-6 CASL IIのアセンブラ命令

| アセンブラ命令 | 意　　味 |
|---|---|
| START | プログラムの名前と実行の開始点を指定する |
| END | プログラムの終了点を指定する |
| DC | 指定されたデータをその位置に置く |
| DS | 指定された語数だけ記憶域を確保する |

### 4.5.1 START命令

本書に付属のCASL IIアセンブラが出力するアセンブラリストを見ると分かるのですが、STARTとENDという命令に対しては、対応する機械語がありません。これらの命令を一般に**擬似命令**と呼び、アセンブラに対して指示を与えるために使います。なお、CASL IIでは、擬似命令とは呼ばずに、アセンブラ命令と呼んでいます。

さて、START命令は、「ここからアセンブラ言語の記述を開始しますよ」ということと「このプログラムの名前はSAMP1ですよ」ということをアセンブラに伝えます。

START命令は、次の構文をしています。

```
プログラム名　START　[実行開始番地]
```

プログラム名は、第1カラムから記述するラベルであり、このプログラムの名前を意味します。これは省略できませんので必ず記述する必要があります。また、[ ]の記号はその中が省略可能であることを意味します。実行開始番地は、実際に、このプログラムの実行が開始されるアドレスをラベルで示します（詳細は後述）。

いくつか例を示しましょう。

```
┌───────── 第1カラム目
↓
PRG1    START
        LD      GR2,A
        ADDA    GR2,B
        ST      GR2,C
        RET
A       DC      1
B       DC      2
C       DS      1
        END
```

上記の例では、START命令のラベルPRG1がこのプログラムの名前になります。この名前は、第12

章で学習するように、他のプログラムからこのプログラムを利用するときに使います。

次の例は、開始番地を指定している例です。

```
PRG2    START   BEGIN
A       DC      1
B       DC      2
C       DS      1
BEGIN   LD      GR2,A
        ADDA    GR2,B
        ST      GR2,C
        RET
        END
```

上記のプログラムでは、このプログラムの実行の開始位置が「BEGINというラベルの付いた機械語から」であることを指定しています。このようにすると、オペレーティングシステムは、プログラムの先頭からではなく、プログラムの途中からこの機械語プログラムを実行します。

また、PRG2というラベルがSTARTに付けられているので、このプログラムの名前は、PRG2となります。

なお、START命令にコメントを書く場合には、注意が必要です。START命令はオペランドの指定が選択できるCASL IIの唯一の命令です。そのため、以下の例に示すように、開始位置を指定した場合、オペランドがあるので、コメントを書くにはセミコロンが不要となります。しかし、開始位置を指定しないとオペランドのない命令となり、コメントを書く場合、セミコロンが必要となります。

```
PRG     START   BEGIN   コメントにセミコロンは不要です
PRG     START   ;コメントにセミコロンが必要です
```

## 演習問題4-5-1

(1) 次の2つのプログラムに付けられた名前と、これらのプログラムが0番地からロードされたときの実行開始アドレスを示してください。

```
PRG3     START
         LD       GR2,A
         ADDA     GR2,B
         ST       GR2,C
         RET
A        DC       1
B        DC       2
C        DS       1
         END

PRG4     START    BEGIN
A        DC       1
B        DC       2
C        DS       1
BEGIN    LD       GR2,A
         ADDA     GR2,B
         ST       GR2,C
         RET
         END
```

(2) 次のSTART命令の記述の誤りを指摘してください。

```
PRGM     START    3
         DC       1
         DC       2
         DC       3
         LAD      GR1,5     ここが3番地になる
```

### 4.5.2 END命令

END命令は、「ここでこのプログラムの記述を終了しますよ」ということをアセンブラに伝えます。したがって、CASL IIのプログラムは必ず、STARTで始まって、ENDで終了します。END命令には、ラベルを付けてはいけないので注意してください。

## ポイント

CASL IIのプログラムは必ず、STARTで始まって、ENDで終了します。

# 演習問題4-5-2

なぜ、END命令にはラベルを付けることができないのでしょう。

### 4.5.3 DC命令

(1) DC命令の構文

DC命令は、次の構文をしていて、そのオペランドに書かれた値をその語に格納するという命令です。

```
[ラベル]  DC  オペランド
```

オペランドには、10進定数、16進定数の他に文字定数およびアドレス定数を書くことができます(表4-7)。また、次のような簡略記法も用意されています。

```
[ラベル]  DC  オペランド1 [, オペランド2],...
```

つまり、必要な個数だけ、オペランドをコンマで区切って並べることができます。上記の表現は、次のように解釈されます。

```
DC  オペランド1
DC  オペランド2
```

↓

表 4-7 DC命令のオペランド

| 定数の種類 | 記述法 | 説　明 |
|---|---|---|
| 10進定数 | n | nは、－32768～32767の数値とする。もし、nがこの範囲外の場合、その下位16ビットが有効となる。 |
| 16進定数 | #h | hは、4桁の16進数(16進数であるから、0～9、A～Fの数字を使う)である。したがって、必ず #0000～#FFFFの間になる。 |
| 文字定数 | '文字列' | 1文字以上の文字を指定する。アポストロフィ(')を書くときは2個連続して書く。たとえば、'this is a message'や'Let''s go'などが文字定数である。データの格納の方法は、4.5.3(4)で示す。 |
| アドレス定数 | ラベル | 他で定義されたラベルを書く。データの格納の方法は、4.5.3(5)で示す。 |

(2) DC命令の意味

たとえば、以前示したプログラムPRG1の機械語を見てください。

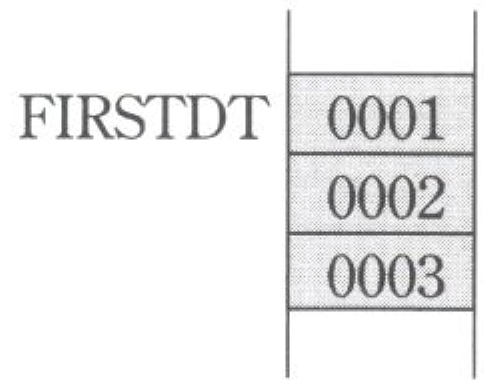
```
        プログラム        開始番地   機械語

PRG1    START
        LD      GR2,A     0          1020 0006
        ADDA    GR2,B     2          2020 0007
        ST      GR2,C     4          1120 0008
        RET               5          8100
A       DC      1         6          0001
B       DC      2         7          0002
C       DS      1         8          8029
        END
```

ラベルAが置かれたアドレス6番地には、DC命令のオペランドで指定された値1が格納されています。ラベルBの部分には2が格納されています。

なお、通常はDC命令にラベルを付けて、その値を利用することが一般的ですが、ラベルは書かなくても構いません。ラベルがなくても、記述された順番で記憶装置に格納されていきます。

```
FIRSTDT   DC   1    ラベル付きのDC命令
          DC   2    ラベル無しのDC命令
          DC   3    ラベル無しのDC命令
```




特に、上記の場合には、次のように記述することもできます。

```
FIRSTDT   DC   1,2,3
```

このとき、「1, 2, 3」の中には決して空白を入れないで下さい。空白を入れるとそれ以降がコメントになってしまいます。以下の例では網掛け部分がコメントとしてアセンブラに認識されてしまいます。

```
FIRSTDT   DC   1,2      ,3
```

(3) オペランドに10進定数・16進定数を記述する場合

DC命令のオペランドに10進定数・16進定数を記述する場合の例を以下に示します。

```
10進数の場合                 置き換わる値(16進)

N1     DC     −123           FF85
N2     DC     12345          3039
N3     DC     −1             FFFF
N4     DC     0              0000
N5     DC     32767          7FFF
N6     DC     −32768         8000
```

| 16進数の場合 | | | 置き換わる値(16進) |
|---|---|---|---|
| H1 | DC | #0000 | 0000 |
| H2 | DC | #0007 | 0007 |
| H3 | DC | #000F | 000F |
| H4 | DC | #0045 | 0045 |
| H5 | DC | #1234 | 1234 |
| H6 | DC | #A0A0 | A0A0 |
| H7 | DC | #FFFF | FFFF |

(4) オペランドに文字定数を置く場合

オペランドに文字定数を置くと、最初の文字の文字コードが第1語の下位8ビットに格納され、次の文字の文字コードが第2語の下位8ビットに格納されます。3つ以上文字がある場合には、以降、その順で下位8ビットにその文字コードが格納されていきます。なお、各語の上位8ビットには0が格納されます。

それでは、具体的な例で示しましょう。

```
MOJI1    DC      'abc'
MOJI2    DC      'XYZ'
```

上記のDC命令によって、次の値が生成されます。

| | | | | |
|---|---|---|---|---|
| MOJI1 | DC | 'abc' | 0061 | 1語目：'a'の文字コード 61(16) |
| | | | 0062 | 2語目：'b'の文字コード 62(16) |
| | | | 0063 | 2語目：'c'の文字コード 63(16) |
| MOJI2 | DC | 'XYZ' | 0058 | 1語目：'X'の文字コード 58(16) |
| | | | 0059 | 2語目：'Y'の文字コード 59(16) |
| | | | 005A | 2語目：'Z'の文字コード 5A(16) |

使用できる文字は、JIS X0201 ローマ字・片仮名用8単位符号(＊1)で規定された文字です。その表を表4-5に示します。ただし、「情報処理技術者出題範囲」の中で規定されている文字は、この表で示されている部分だけで、それ以外の文字(を規定するビット構成)が必要な場合には、問題文中で与えることになっています。

アポストロフィ(')を書くときは2個連続して書きますので、She doesn't like itは次のように指定します。

```
MES      DC      'She doesn''t like it'
```

(5) オペランドにアドレス定数を置く場合

DC命令のオペランドには、ラベルを書くこともできます。次のプログラムを見てください。

---

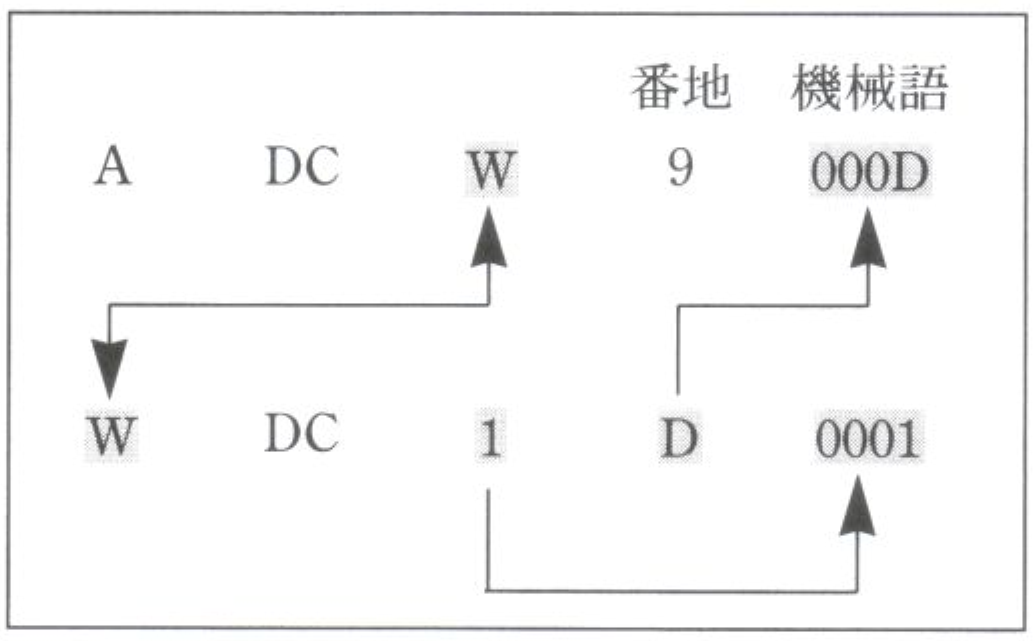
(＊1) 日本工業規格(JIS)で定められた文字コードの規格です。

```
ADRC     START
         LD       GR1,A
         LD       GR2,B
         LD       GR3,C
         LD       GR4,D
         RET
A        DC       W          Wはアドレス定数
B        DC       X          Xはアドレス定数
C        DC       Y          Yはアドレス定数
D        DC       Z          Zはアドレス定数
W        DC       1
X        DC       2
Y        DC       3
Z        DC       4
         END
```

これをCASL IIアセンブラにかけると次のような機械語が生成されます。

```
アセンブラ言語                番地      機械語

ADRC     START
         LD       GR1,A       0       1010   0009
         LD       GR2,B       2       1020   000A
         LD       GR3,C       4       1030   000B
         LD       GR4,D       6       1040   000C
         RET                  8       8100
A        DC       W           9       000D
B        DC       X           A       000E
C        DC       Y           B       000F
D        DC       Z           C       0010
W        DC       1           D       0001
X        DC       2           E       0002
Y        DC       3           F       0003
Z        DC       4          10       0004
         END
```




W, X, Y, Zはそれぞれ$D_{(16)}$番地、$E_{(16)}$番地、$F_{(16)}$番地、$10_{(16)}$番地に配置されていることをまず記憶してください。そして、Aの場所にはWの配置場所のアドレスである$000D_{(16)}$が、Bの場所にはXの配置場所のアドレスである$000E_{(16)}$が、Cの場所にはYの配置場所のアドレスである$000F_{(16)}$が、Dの場所にはZの配置場所のアドレスである$0010_{(16)}$が、それぞれ格納されています。このアドレスのことを**アドレス定数**と呼びます。

このプログラムを実行すると、ラベルA, B, C, Dの内容が、それぞれ、レジスタ GR1, GR2, GR3, GR4 にロードされます。したがって、レジスタGR1, GR2, GR3, GR4には、Wのアドレス、Xのアドレス、Yのアドレス、Zのアドレスが格納されます。

他のラベルのアドレスを知ることによって、どのようなプログラムが書けるかは、12.1.4項で説明します。

## 演習問題4-5-3

次のDC命令によって、確保される語数を求めてください。

(1) DC　2, 3, 4, 5
(2) DC　'LET''S GO'
(3) DC　ADR1, ADR2, ADR3　(ADR1,ADR2,ADR3はアドレス定数とします)

### 4.5.4 DS命令

DS命令は、次の構文をしています。

```
[ラベル]　DS　語数
```

語数の部分には、10進定数で数値を指定します。16進定数では指定できないので注意してください。

DS命令は、そのアドレスから、語数で示された語数分だけデータを確保するという命令です。たとえば、次の例を見てください。

```
A    DC    5
B    DS    3
C    DC    4
```

これを機械語に変換すると次のようになります。

```
A    DC    5    0005
B    DS    3    0000   ↑
                0000   3語分確保される
                0000   ↓
C    DC    4    0004
```

この場合、ラベルBの「DS 3」によって、そこから3語が確保されます。

この例では、各語には0000が格納されていますが、「仕様」では、そこの値には何の規定もありませんので、0000が入っていることを仮定することはできません。そこで、本書のCASLDVのCASL IIアセンブラでは、DS命令で確保される語の値を0000として機械語に変換することもできますし、適当な値を設定する(CASLDVの標準値)こともできます。

DC命令は、値を設定したい場合に使いますが、このDS命令は、計算結果を保持する領域を確保するときに使います。

特に、DS命令を使うと、多くの語を連続して確保することができるので、第11章で学習する表を扱うときに便利です。

なお、DS命令の語数にはゼロを指定することもできます。その場合、そのラベルだけが定義されてデータは確保されません。以下の例ではXはYと同じアドレスになり、GR1には5がロードされます。

```
        LD      GR1,X
         :
X       DS      0
Y       DC      5
```

初心者のうちは、DC命令とDS命令の使い方の違いが分かりにくいと思いますので、例を示して説明します。次の例は、A+B→Cを計算するプログラムです。

```
SAMPDS  START
        LD      GR1,A
        ADDA    GR1,B
        ST      GR1,C
        RET
A       DC      5
B       DC      3
C       DS      1           領域が1語分確保される
        END
```

AとBには、それぞれ5と3という値を設定したいので、「A DC 5」、「B DC 3」というようにDC命令を使います。しかし、Cは、そこに最初に入っている値はまったく重要ではなく、計算した値をそこに格納するために必要な領域です。つまり、「C　DS　1」として単にCの領域として1語分確保するだけでよいのです。

しかし、このプログラムの場合は、Cの領域にあらかじめデータが入っていたとしても、「ST GR1,C」によってレジスタGR1の内容がCにストア(複写)されます。そこで、「C　DC　0」としている次のプログラムでも、特に問題はありません。つまり、最終的には、領域CにA+Bの値が格納されればよいのですから。

```
SAMPDS1  START
         LD      GR1,A
         ADDA    GR1,B
         ST      GR1,C
         RET
A        DC      5
B        DC      3
C        DC      0          領域Cに0がセットされる
         END
```

つまり、値を設定せずに1語だけ確保して使う方法では、DS命令でもDC命令でも利用面では、それほど大きな違いはありません。DS命令が効果的な場面は、第11章の表を扱うときです。また、4.6節で説明するIN命令、OUT命令を用いるときも便利です。

## 演習問題4-5-4

次のDS命令の記述の誤りを指摘してください。

(1) DS　'This is a pen.'

(2) DS　#00AA

(3) DS　−34

# 4.6 マクロ命令

**マクロ命令**(macro instruction)とは、その命令を他の命令に置き換える命令で、CASL IIでは、IN, OUT, RPUSH, RPOPの4つだけが定義されています(*1)。これらの命令が、置き換わる命令についての具体的な記述は、「仕様」では定義されていませんが、「情報処理技術者試験　出題範囲」「参考資料」の中で、INマクロ命令が次のように置き換わることを想定しているという記述があります。たとえば、

```
LABEL      IN      IBUF,LEN
```

というマクロ命令は、次の機械語に置き換えられます。

```
LABEL   PUSH    0,GR1
        PUSH    0,GR2
        LAD     GR1,IBUF
        LAD     GR2,LEN
        SVC     1
        POP     GR2
        POP     GR1
```

見たこともない命令が続いていますが、この命令によって入力作業を実現します。入力・出力の具体的なプログラミングの仕方については、第7章以降で説明します。ここでは、その概要について説明します。

なお、IN命令あるいはOUT命令の実行後、汎用レジスタ(GR0〜GR7)の値は保たれますが、フラグレジスタの値は変更されます。

## 4.6.1 IN命令

IN命令は、文字データを入力装置から入力するための命令です。IN命令は、次の構文をしています。

| [ラベル]　IN　入力領域, 入力文字長領域 |
|---|

入力領域は、実際にデータを入力する領域のラベルを指定します。したがって、通常は複数文字を入力するので、DS命令を使って、入力領域、入力文字長領域を定義します。IN命令では、最長256文字までの文字データを一度に格納できます。IN命令では、後で例に示すように、1文字を1語に格納します。したがって、この領域は、最大256語分必要となります。入力される文字数があらかじめ分からない場合には256文字分用意しておいたほうがよいでしょう。

また、入力文字長領域には、ラベルを指定します。このラベルの領域には、入力された文字数(0〜256)が格納されるので、1語あれば十分です。

さて、たとえば、次のように領域が定義されていたとしましょう。

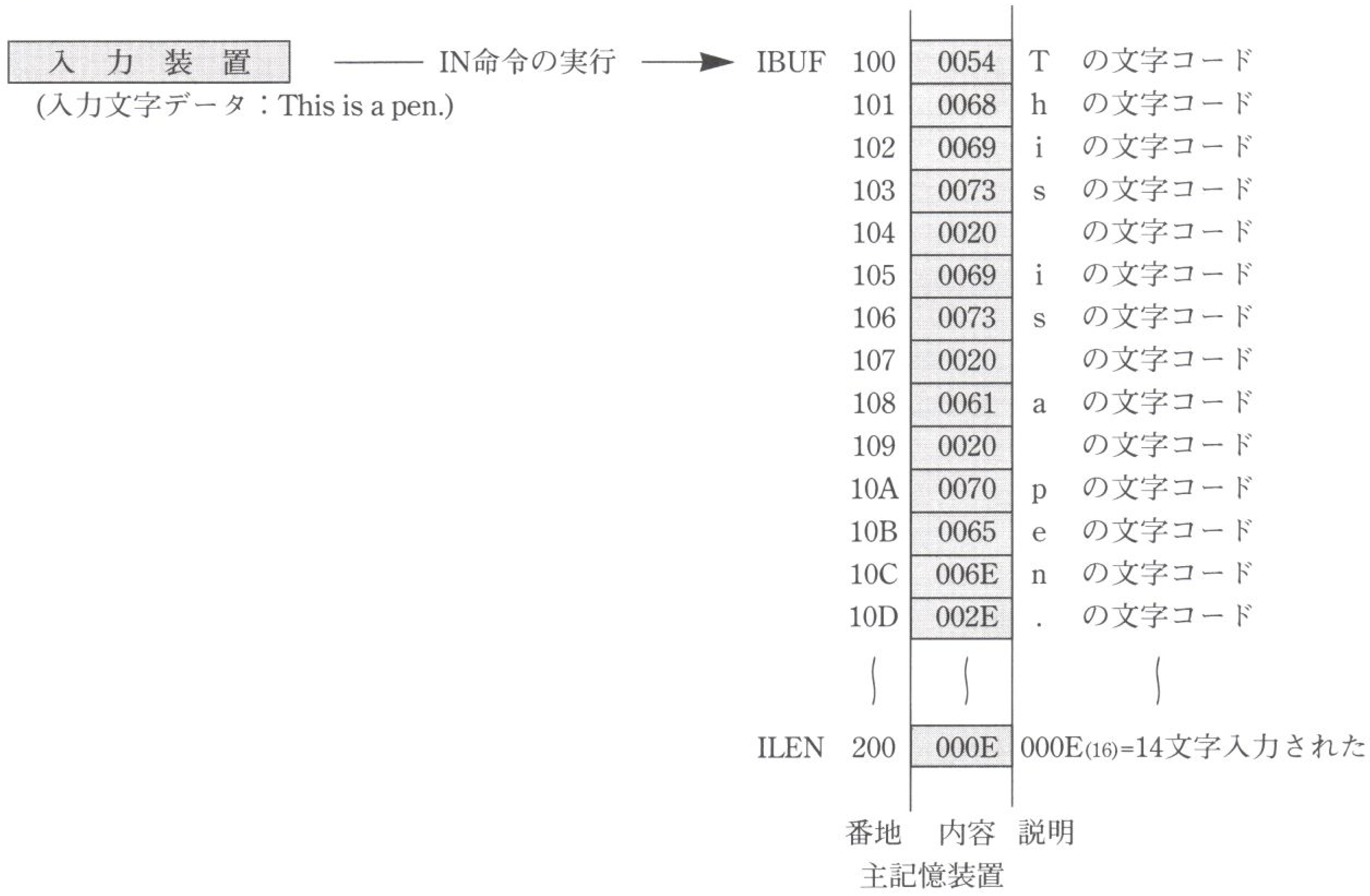
```
IBUF    DS      256◀── 入力領域
ILEN    DS      1  ◀── 入力文字長領域
```

(＊1) 一般のコンピュータには入出力のための機械語命令がありますが、単純化されたコンピュータであるCOMET IIには入出力のための機械語命令が定義されていません。そのため、入出力はここで紹介するマクロを使うことになっています。

今、仮にIBUFが$100_{(16)}$番地から割り当てられたとしましょう。すると、ILENは$200_{(16)}$番地に割り当てられます。このとき、次のIN命令を書くことができます。

```
        IN      IBUF,ILEN
```

IN命令を実行すると、入力装置から文字データを受け取って、それらを主記憶装置に格納します。次の例は、入力装置から、文字データ「This is a pen.」を受け取って、それを$100_{(16)}$番地以降に格納した例です。




この例のように、文字データがIN命令で入力された場合、先頭から1語(下位8ビット)に1文字ずつ格納されていきます。このとき、上位8ビットには0がセットされます。また、ILENの場所($200_{(16)}$番地)には、入力された文字数($14=000E_{(16)}$)が格納されます。

入力装置に送られるデータ(この例では、「This is a pen.」)は、キーボードからの入力であったりファイルからの入力[*1]であったりします。どの装置から入力するかは、このプログラムの実行に先立ってオペレーティングシステムで決めます。本書に付属のCASLDV(COMET IIシミュレータ)では、標準は、入力がキーボードから、出力がディスプレイになっていますが、プログラムの実行に先立ってファイルに変更することができます。

なお、IN命令を実行したときに入力データがない場合（ファイルの終わり（end of file）を検出した場合)、入力文字長領域に－1が格納されます。

### 4.6.2 OUT命令

OUT命令は、記憶装置上の文字データを出力装置から出力するための命令です。OUT命令は、次の構文をしています。

(＊1) すなわち、ファイルの内容をデータとして読み込むことです。

| [ラベル] | OUT | 出力領域,出力文字長領域 |
|---|---|---|

出力領域は、実際にデータを出力する領域のラベルを指定します。したがって、通常複数文字を出力するので、DS命令を使って、出力領域、出力文字長領域を定義します。OUT命令では、最長65535文字までの文字データを一度に出力できます。一度に出力された出力文字データは、1レコード(*1)として扱われます。

OUT命令では、後で例に示すように、1語に格納された下位8ビットを1文字として出力します。したがって、上位8ビットは無視されますので何があってもかまいません。

また、出力文字長領域には、ラベルを指定します。このラベルの領域には、出力すべき文字数を格納しておくので、1語あれば十分です。

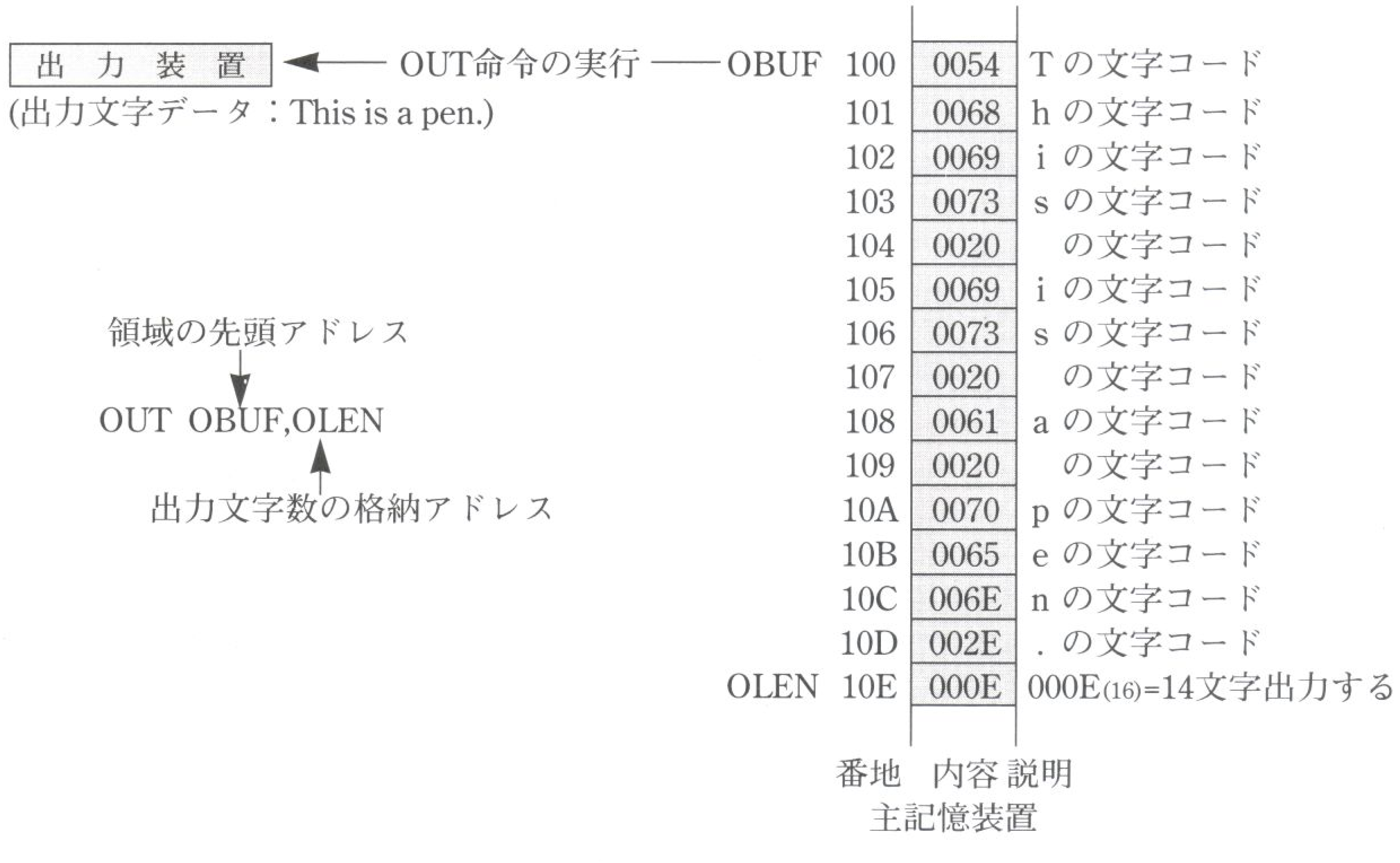
たとえば、次のように領域が定義されていたとしましょう。

```
OBUF     DC       'This is a pen.'        出力領域
OLEN     DC       14          出力文字長領域
```

このとき、次のOUT命令を書くことができます。

```
         OUT      OBUF,OLEN
```

今、仮にOBUFが100(16)番地から割り当てられたとしましょう。すると、OLENは10E(16)番地に割り当てられます。OUT命令を実行すると、主記憶装置内の文字データを出力装置に転送します。次の例は、100(16)番地以降の主記憶装置から、文字データ「This is a pen.」を出力装置に転送した例です。




(＊1)レコードとは、入出力の単位のことです。COMET IIの仕様では、出力の場合、16ビットで表現できる長さ(65535)までのデータを出力できるようになっていますが、実際には、レコードの長さの制限があります。本書に付属のCASLDV(COMET IIシミュレータ)では、1レコードの長さを1024文字までに制限しています。

この例では、「DC 'This is a pen.'」でセットされた領域を出力対象としているので、それらの上位8ビットには0が格納されていますが、この部分は何であっても構いません。

出力装置に送られるデータ(この例では、「This is a pen.」)は、実際にはディスプレイに表示されたり、ファイルに格納されたりします。どこに出力するかは、このプログラムの実行に先立ってオペレーティングシステムで決めます。本書に付属のCASLDV(COMET IIシミュレータ)では、標準は、出力がディスプレイになっていますが、プログラムの実行に先立ってファイルに変更することができます。

※RPUSH命令は、汎用レジスタ（GR1, GR2, …, GR7）の内容を、この順でスタックにPUSHします。

```
[ラベル]    RPUSH
```

RPOP命令は、スタックからデータを順次取り出し、GR7, GR6, …, GR1の順で格納します。

```
[ラベル]    RPOP
```

# 総合問題4

**[演習4-1]** 次のプログラムを、自分の力で手作業で機械語に変換してください。なお、この作業を**ハンドアセンブル**と呼びます。

```
SAMP4   START
        LD      GR1,A
        ADDA    GR1,B
        ST      GR1,C
        RET
A       DC      5
B       DC      3
C       DS      1
        END
```

**[演習4-2]** アセンブラ言語の書き方について次の文章の正誤を判定してください。

(1) ラベルは第2カラムから書くことができる。
(2) ラベル名の長さに制限はない。
(3) GR3はラベル名としては使えないがENDはラベル名として使える。
(4) DC命令には必ずラベルを付けなくてはいけない。
(5) START命令にはラベルを付けないこともある。
(6) 「DS　8,2」は10語分確保する。
(7) CASL IIのプログラムはSTARTで始まりENDで終わる。
(8) 「DC　#FFFF」と「DC　−1」は全く同じ結果になる。
(9) コメントは必ず;(セミコロン)で始まる。
(10) 「DC　'ab'c'」という記述は正しい。

# 第5章 基本プログラミング編

本章では、CASL IIアセンブラの実際のプログラミング例を通して、具体的なアセンブラ言語のプログラミングの方法を学習していきます。

## 5.1 簡単なプログラミング

### 5.1.1 W ← X＋Y－Z

X＋Y－Zの計算を行って、その結果をWに保存するプログラムを示します。まず最初にXの値をレジスタGR0(＊1)にロード(LD)し、GR0にYの値を加えて(ADDA)、さらにZの値を減じれば(SUBA)、GR0にはX＋Y－Zの計算結果が格納されていることになります。次に、GR0の値をWにストア(ST)すれば、WにはX＋Y－Zの計算結果が格納されることになります。そのプログラム(S1.casl)を以下に示します。

```
S1      START
        LD      GR0,X
        ADDA    GR0,Y
        SUBA    GR0,Z
        ST      GR0,W
        RET
X       DC      90
Y       DC      87
Z       DC      77
W       DS      1
        END
```

なお、上記のプログラムは短いので、ちょっと見ただけでも何をしているのかすぐに分かります。しかし、長いプログラムや複雑なプログラムになると、プログラムを読んだだけでは何をしているのか全くわからなくなります。そこで、コメント(comment、4.3節参照)を付けてプログラムの説明を行って、プログラムの意味がよくわかるようにしましょう。

第4章で説明したように、オペランドのある命令では、その命令の後に書いた文字は、コメントとして扱われます。また、オペランドのない命令の場合、その後にセミコロン(;)を書くと、その後はコメントとして扱われます。あるいは、何も命令ない行の場合、セミコロンを最初に書いてコメントを書くこともできます。これを注釈行といいますが、セミコロンの前に空白を入れてもかまいません。次のプログラム(S1C.casl)では、最初の3行が注釈行になります。

(＊1) たまたま、GR0を選んだだけで、この場合にはGR1, GR2, GR3, GR4, GR5, GR6, GR7のどのレジスタでも構いません。

```
;以下の計算を行うCASLプログラム
;         W=X+Y−Z
;
S1C       START           ; プログラムの記述開始(プログラム名 S1)
          LD      GR0,X         GR0 ← X
          ADDA    GR0,Y         GR0 ← GR0 + Y
          SUBA    GR0,Z         GR0 ← GR0 − Z
          ST      GR0,W         GR0 → W
          RET
X         DC      90      Xの値を90に設定
Y         DC      87      Yの値を87に設定
Z         DC      77      Zの値を77に設定
W         DS      1       Wの領域を1語分確保
          END             ; プログラムの記述終了
```

この例では、START命令とEND命令にオペランドがないので、セミコロンを付けて、注釈を記述しています。さて、このプログラムを実際にCASLDVを用いて、CASL IIアセンブラにかけて、COMET IIシミュレータで実行してみましょう。具体的な方法は、CASLDVのマニュアル（本書中及びCD-ROM中）を参照してください。ここでは、Unix(*1) およびWindowsで共通に使えるコマンドライン版を用いた場合の簡単な流れだけを説明します（図5-1参照）。

まず、上記のプログラムをエディタを使って作成します。それをCASL IIアセンブラにかけて、機械語に変換します。アセンブラは、取り合えず先頭の命令が0番地であると解釈してアセンブルします。その様子を示すために、CASL IIアセンブラの出力したアセンブラリストを示します(*2)。**アセンブラリスト**とは、機械語との関連が分かりやすくなるように元のプログラムを編集して出力したものです。なお、CASLDVにはエディタが内蔵され、「アセンブラ・実行」がボタン１つで行えます。

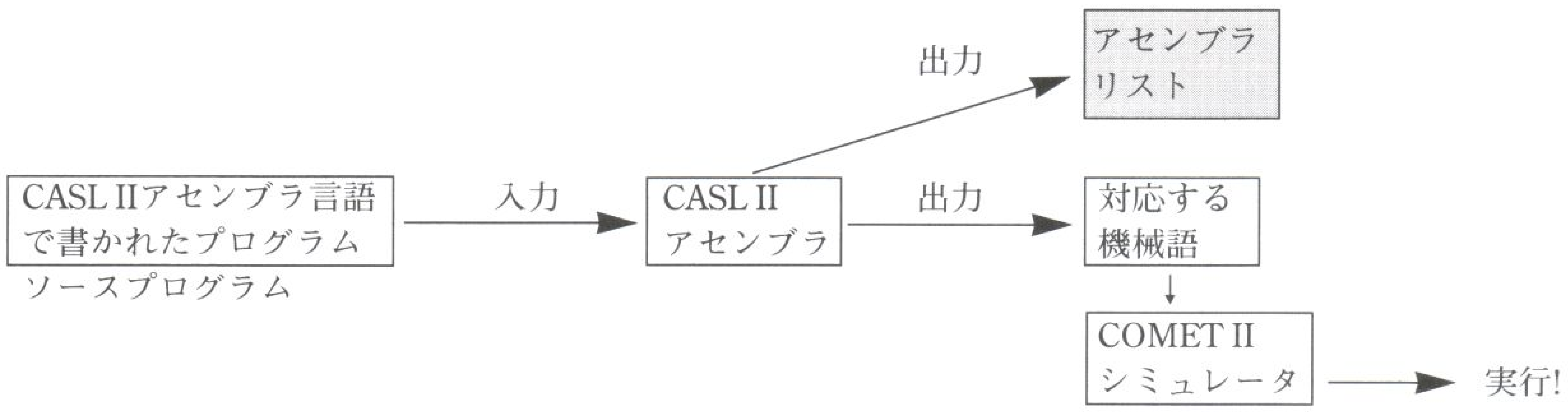



**図 5-1　アセンブラ処理の流れとアセンブラリスト**

この元のプログラムを**ソースプログラム**(source program)と呼びます。さて、このプログラムリストの左側に、機械語（machine language）が示されています。これはちょうど1行に1語ずつ配置してありますので、機械語が主記憶装置へ配置されたイメージの形式になっています。

Adrには、先頭を0番地としてその機械語のアドレスが16進数で示されています。このアドレスは、

(＊1) Unix(Linux, FreeBSDなど)版は、本書に付属のCD-ROMには入っておりません。必要な方は序文をご覧ください。
(＊2) CASLDVの場合には、アセンブラリストは画面左上のペイン(ウインドウ)に表示され、印刷することもできます。しかし、次ページに示すアセンブラリストをファイルに出力することもできます。メニューから「オプション」を選び、「アセンブラオプションの指定」を選びます。そして、表示されたダイアログの「アセンブラリストをファイルに出力する」をチェックします。すると、アセンブラにかけたときに、ソースファイルのあるフォルダに「〜.LST」という名前のファイルが作成され、その中にアセンブラリストが格納されます。

この機械語プログラムが0番地からロードされた場合には、ロード先のアドレスと一致します。しかし、別の番地からロードされた時は、その先頭アドレスからの相対アドレスとなります。

Codeの部分には、機械語が16進数で示されています。この部分が空欄ということは、その行に対応する機械語が生成されないということを意味します。

Line以降は、ソースプログラムが示されています。Lineには、そのソースプログラムの行番号が示されています。この行番号は、ところどころ、飛び飛びになっています。これは、その行の機械語が2語になっている場合です（LD命令、ADDA命令、SUBA命令、ST命令など）。

Programの部分には、ソースプログラムそのものが示されています。

以下にコマンドライン版のCASL IIアセンブラによって生成されたS1C.caslのプログラムリストを示します。

```
   Machine       Source List (S1C.casl)
 Adr.  Code    Line   Program
      |      |   1 | ; 以下の計算を行うCASLプログラム
      |      |   2 | ;        W = X + Y - Z
      |      |   3 | ;
      |      |   4 |S1C     START           ; プログラムの記述開始(プログラム名 S1)
   0  | 1000 |   5 |        LD    GR0,X           GR0←  X
   1  | 0009 |     |
   2  | 2000 |   6 |        ADDA  GR0,Y           GR0←  GR0 +  Y
   3  | 000A |     |
   4  | 2100 |   7 |        SUBA  GR0,Z           GR0←  GR0 -  Z
   5  | 000B |     |
   6  | 1100 |   8 |        ST    GR0,W           GR0→  W
   7  | 000C |     |
   8  | 8100 |   9 |        RET
   9  | 005A |  10 | X      DC    90    Xの値を90に設定
   A  | 0057 |  11 | Y      DC    87    Yの値を87に設定
   B  | 004D |  12 | Z      DC    77    Zの値を77に設定
   C  | 8029 |  13 | W      DS     1    Wの領域を1語分確保
      |      |  14 |        END         ; プログラムの記述終了
   ↑     ↑      ↑           ↑
  番地  機械語  行番号      プログラム
(アドレス)
```

このプログラムを実行させるためには、主記憶装置へこのプログラムを転送しなければなりません。3.3節、3.4節で学習したように、この作業もロードと呼び、この主作業を行うプログラムのことを**ローダ**と呼びます。前述のように、もし、ロードを0番地から行えば、ロードされた機械語とアセンブラリストに示されているアドレスはまったく同じになります。一方、CASLDV（GUI版）で表示されるアセンブラリスト(画面左上のペイン)はロードされたアドレスになっています。ここでは、説明のため、0番地からロードして実行したときの動作結果を示します。

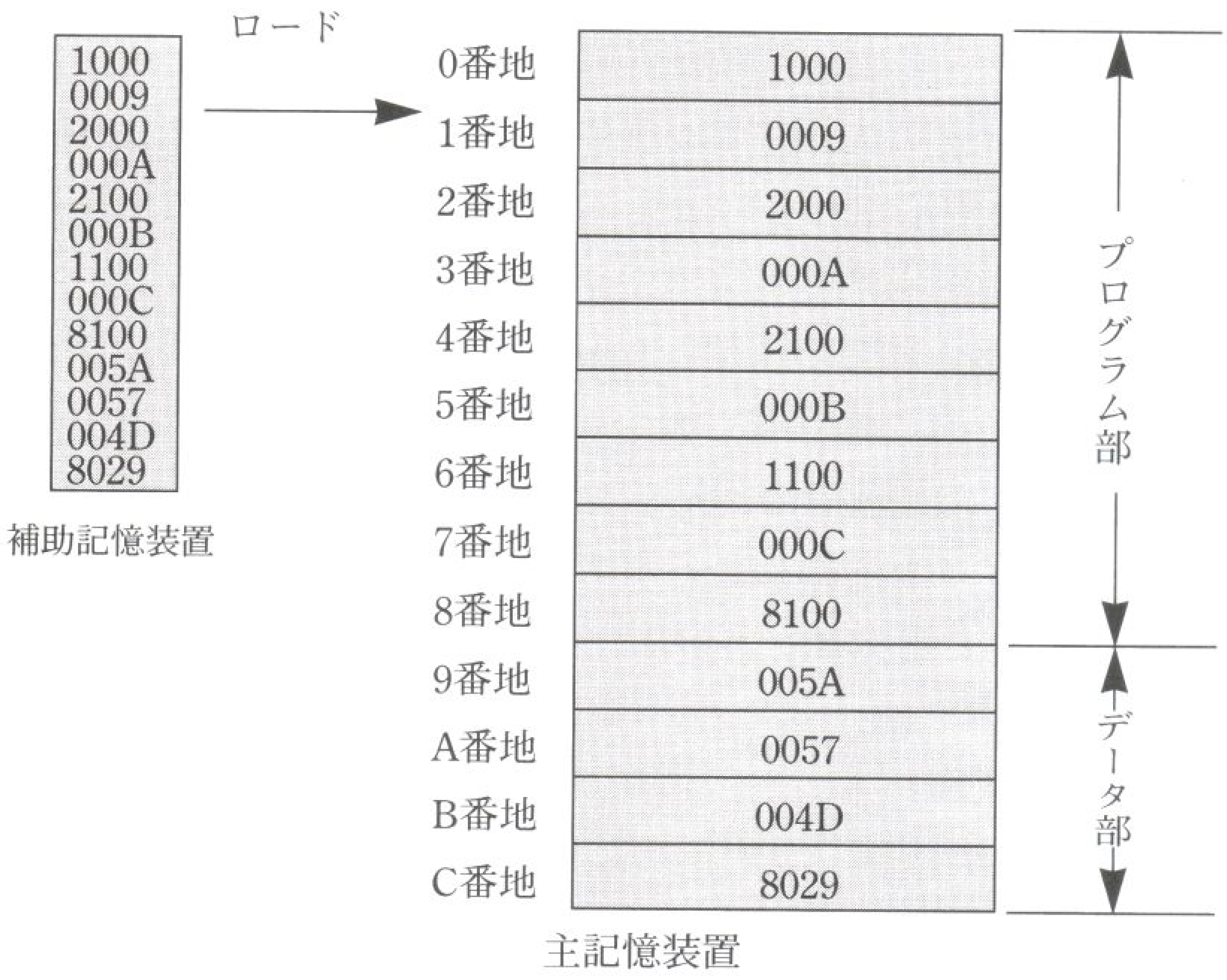



**図5-2 ローダの様子：0番地目からロードした例**

このプログラムの具体的な実行の様子を示します。まず、0番地（および1番地）の命令が実行され、9番地の内容(90＝$5A_{(16)}$)がレジスタGR0にロードされます。GR0の値は$5A_{(16)}$になります。次に、2番地（および3番地）の命令が実行され、A番地の内容(これがラベルYの値で87＝$57_{(16)}$)がレジスタGR0に加えられます。すると、GR0の値は、90＋87＝$5A_{(16)}$＋$57_{(16)}$ですから、177＝$B1_{(16)}$となります。次に、4番地（および5番地）の命令が実行され、レジスタGR0からB番地の内容(これがラベルZの値で77＝$4D_{(16)}$)を引きます。すると、GR0の値は、177－77＝$B1_{(16)}$－$4D_{(16)}$ですから、100＝$64_{(16)}$になります。この値は、X＋Y－Z＝90＋87－77＝100を意味している訳です。次に6番地（および7番地）の命令が実行され、GR0の値がC番地の内容（ラベルWのこと）に格納されます。最後に8番地の命令が実行されてプログラムの実行が終了します。

それでは、CASLDV(COMET IIシミュレータ)を使って、このプログラムを実際に実行して、機械語の動作を観察してみましょう。以下に、プログラムS1Cの実行結果を示します(*1)。なお、このような動作の様子の出力を**トレース**(trace)と呼ぶことがあります。

```
 PR  Instruction          SP   OSZ  GR0  GR1  GR2  GR3  GR4  GR5  GR6  GR7  Explanation
0000 LD     GR0,0009     7530  ---  005A C823 98BE E784 CAE1 BD6C ACD6 F2AE 0009:[h:005A, s:90, u:90]
0002 ADDA   GR0,000A     7530  ---  00B1 C823 98BE E784 CAE1 BD6C ACD6 F2AE 000A:[h:0057, s:87] +=> GR0:[s:177]
0004 SUBA   GR0,000B     7530  ---  0064 C823 98BE E784 CAE1 BD6C ACD6 F2AE 000B:[h:004D, s:77] -=> GR0:[s:100]
0006 ST     GR0,000C     7530  ---  0064 C823 98BE E784 CAE1 BD6C ACD6 F2AE GR0:[h:0064, s:100, u:100]
0008 RET                 7530  ---  0064 C823 98BE E784 CAE1 BD6C ACD6 F2AE Return to OS
```

PRの下には、実際にその命令が格納されているアドレス、Instructionの下には、その命令のニーモニックコード（ただし、ラベルのアドレスは実際にロードされたアドレスに変換されています）、SPにはスタックポインタの値(この例では使っていません)、OSZにはフラグレジスタの各ビットOF, SF, ZFの値が示されています。この例ではフラグレジスタのどのビットも1になっていません。－という

(*1) この実行結果は、コマンドライン版のときはcomet_IIコマンドを起動すると標準出力に出力され、GUI版であるCASLDVでは最も下のペイン(表示領域)に表示されます。

表示はそのフラグが0であることを示しています。もし、フラグが1であれば、OFならO、SFならS、ZFならZが表示されます。GR0～GR7の下には、そのレジスタの値が示されています。この例を見ると、最初、GR1～GR7には変な値が入っています。レジスタの初期値は、最初、どのような値が入っているか分かりません。このプログラムでは、最初、レジスタGR0に9番地の値を格納しているので、GR0だけは意味のある値(90＝5A(16))が入っていますが、それ以外の汎用レジスタの値は無意味なものです。

プログラムでは、汎用レジスタに値を設定するまでは無意味な値が入っているので、その汎用レジスタの値を使ってはなりません。しかし、仮にその無意味な値を使ったとしても、エラーとしては扱われず、無意味な値での計算が行われるだけです(*1)。

Explanationには、命令の動作の説明が示されています。「LD GR0,0009」命令の場合には、0009番地の値がGR0にロードされるわけですから、9番地の内容が0009:[h:005A,s:90,u:90]という形式で示されています。hが16進数、sが符号付き10進数、uが符号無し10進数を意味します。

「ADDA GR0,000A」命令の場合には、まず、加算対象となるA番地の内容が示され、それが加えられた後のレジスタの値が示されています。A番地の内容には、sすなわち符号付きの値しか示されていません。それは、ADDAという命令が符号付きの10進数だけを対象とする命令であるからです。符号無しの値は、COMET IIでは、論理データとして扱われ、それに対応する加算命令としてADDLという命令がありますが、それは、6.2節で扱います。SUBAも同様に符号付きデータを減算する命令です。符号付きデータと符号無しデータの計算の具体的な違いについては、2.6節で解説しています。

最終的に、「ST GR0,000C」でGR0の内容がWに保存されたことになります。

ところで、この例では0番地からロードしましたが、一般に主記憶装置の先頭部分(0番地以降)は、システムが利用していることが多く、ユーザはそれ以外の部分にロードしなければならないことが多いのです。そこで、本書に付属のGUI版であるCASLDV(COMET IIシミュレータ)は任意のアドレスに機械語のプログラムをロードできます(*2)。次の例は、F000番地以降にロードして実行したときの結果です。

```
  PR  Instruction          SP    OSZ   GR0   GR1   GR2   GR3   GR4   GR5   GR6   GR7  Explanation
F000  LD       GR0,F009    7530  ---   005A  C823  98BE  E784  CAE1  BD6C  ACD6  F2AE F009:[h:005A, s:90, u:90]
F002  ADDA     GR0,F00A    7530  ---   00B1  C823  98BE  E784  CAE1  BD6C  ACD6  F2AE F00A:[h:0057, s:87] +=> GR0:[s:177]
F004  SUBA     GR0,F00B    7530  ---   0064  C823  98BE  E784  CAE1  BD6C  ACD6  F2AE F00B:[h:004D, s:77] -=> GR0:[s:100]
F006  ST       GR0,F00C    7530  ---   0064  C823  98BE  E784  CAE1  BD6C  ACD6  F2AE GR0:[h:0064, s:100, u:100]
F008  RET                  7530  ---   0064  C823  98BE  E784  CAE1  BD6C  ACD6  F2AE Return to OS
```

命令の配置されているアドレスもずれていますが、各命令中のアドレスもすべてずれています。たとえば、最初のLD命令は、「LD GR0,F009」となり、F009(16)番地の内容をレジスタGR0にロードするようになっています。これは、命令全体がずれたため、データ90が入っている場所(つまり、ラベルXの領域)もずれてF009番地となったのです。このように、機械語プログラムはそれがロードされる開始アドレスによって、実際のアドレスの位置が異なります。このアドレスの補正計算は、ローダが行ってくれるので、ラベルを用いている限り、私たちが気にする必要はありません。

(*1) 実際のコンピュータでは、初期化せずに汎用レジスタに無意味な値を入れたまま、その値を計算に使ってもエラーが起こりませんが(実はチェックできない)。本書に付属のCOMET IIシミュレータは、初心者教育用なので、初期化していない汎用レジスタの値を利用しようとするとエラーを出すようにしています（このチェックをはずすこともできます)。

(*2) メニューから「オプション」→「実行時オプションの指定」を選び、「ロードする機械語の先頭アドレス」を変更する。

```
S1     START
       LD       GR0,X
       ADDA     GR0,Y
       SUBA     GR0,Z
       ST       GR0,W
       RET
X      DC       90
Y      DC       87
Z      DC       77
W      DS        1
       END
```

0000番地からロード

```
Adr. Code ニーモニック
0000 1000 LD       GR0,0009
0001 0009
0002 2000 ADDA     GR0,000A
0003 000A
0004 2100 SUBA     GR0,000B
0005 000B
0006 1100 ST       GR0,000C
0007 000C
0008 8100 RET
0009 005A DC   90
000A 0057 DC   87
000B 004D DC   77
000C 8029 DS    1
```

F000番地からロード

```
Adr. Code ニーモニック
F000 1000 LD       GR0,F009
F001 F009
F002 2000 ADDA     GR0,F00A
F003 F00A
F004 2100 SUBA     GR0,F00B
F005 F00B
F006 1100 ST       GR0,F00C
F007 F00C
F008 8100 RET
F009 005A DC   90
F00A 0057 DC   87
F00B 004D DC   77
F00C 8029 DS    1
```

**図5-3 異なるアドレスにロード**

しかし、このようにローダが、アドレスを管理してくれるのは、あくまでもラベルで生成されたアドレスに対してだけです。

次のプログラムを見てください。次のプログラム（S1H.casl）は、ラベルXの領域が0009(16)番地で始まることを利用して、直接16進定数で、レジスタにロードするアドレスを指定している例です。

```
S1H     START              ; プログラムの記述開始(プログラム名 S1H)
        LD      GR0,#0009    GR0 ← X
        ADDA    GR0,#000A    GR0 ← GR0 ＋ Y
        SUBA    GR0,#000B    GR0 ← GR0 － Z
        ST      GR0,#000C    GR0 → W
        RET
X       DC      90         Xの値を90に設定
Y       DC      87         Yの値を87に設定
Z       DC      77         Zの値を77に設定
W       DS      1          Wの領域を1語分確保
        END                ; プログラムの記述終了
```

このプログラムは、もちろん正しくアセンブルされます。そして、0番地からロードして実行すると次のようになります。

| PR | Instruction | | SP | OSZ | GR0 | GR1 | GR2 | GR3 | GR4 | GR5 | GR6 | GR7 | Explanation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | LD | GR0,0009 | 7530 | --- | 005A | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 0009:[h:005A, s:90, u:90] |
| 0002 | ADDA | GR0,000A | 7530 | --- | 00B1 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 000A:[h:0057, s:87] +=> GR0:[s:177] |
| 0004 | SUBA | GR0,000B | 7530 | --- | 0064 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 000B:[h:004D, s:77] -=> GR0:[s:100] |
| 0006 | ST | GR0,000C | 7530 | --- | 0064 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | GR0:[h:0064, s:100, u:100] |
| 0008 | RET | | 7530 | --- | 0064 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | Return to OS |

領域Wのアドレスにちゃんと100が格納されているので、うまく動いていることが分かります。しかし、このプログラムを1000(16)番地にロードして、そこから実行してみると、次のようになります。

| PR | Instruction | | SP | OSZ | GR0 | GR1 | GR2 | GR3 | GR4 | GR5 | GR6 | GR7 | Explanation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | LD | GR0,0009 | 7530 | --Z | 0000 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 0009:[h:0000, s:0, u:0] |
| 1002 | ADDA | GR0,000A | 7530 | --Z | 0000 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 000A:[h:0000, s:0] +=> GR0:[s:0] |
| 1004 | SUBA | GR0,000B | 7530 | --Z | 0000 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 000B:[h:0000, s:0] -=> GR0:[s:0] |
| 1006 | ST | GR0,000C | 7530 | --Z | 0000 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | GR0:[h:0000, s:0, u:0] |
| 1008 | RET | | 7530 | --Z | 0000 | C823 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | Return to OS |

プログラムを1000(16)番地からロードしているので、領域Aのアドレスは1009(16)番地になっているはずですが、アセンブラ言語の記述が「LD GR0,#0009」となっているため、あくまでも9番地の内容をGR0にロードしてしてしまったのです。そこには、この場合では、無意味な0という数が入っていたようです(＊1)。

この例から分かることは、プログラム中に直接アドレスを書くと、それは、ずれることのない、そのままの番地として扱われるということです。このようなアドレスを**絶対アドレス**(absolute address)と呼びます。

一方、ラベルを使ってアドレスを表現する方法では、それがロードされるたびごとにずれます。このようなアドレスを**相対アドレス**(relative address)と呼びます。

## 演習問題5-1-1

(1) W←X－Y＋Zの計算を行うプログラムを作りなさい。

(2) W←X－Y－Zの計算を行うプログラムを作りなさい。

### 5.1.2 D ← 3×(A＋B)－C

本項では、3×(A＋B)－Cを求めて、その結果をDに代入するプログラムを考えましょう。この例では、3倍の計算があります。乗算を行う機械語は、COMET IIにはないので、別の方法で計算しなければなりません。乗算は後述するループやビット演算を用いる方法が一般的ですが、ここでは、(A＋B)を単に3回加えて3×(A＋B)を求めることにしましょう。

(＊1) 本書に付属のCASLDV(comet_IIシミュレータ)は、プログラムで正しく設定していない領域からデータをロードすると実行時エラーとなります。もちろん、その機能は外すことができます。この例は、その機能を外して実行した例です。この機能を外すには、CASLDVの場合、メニューから「オプション」→「実行時のオプションの指定」を選択し、「さまざまな実行時オプション」グループの中から「記憶領域の境界のチェックを行う」チェックマークを外してください。コマンドライン版をお使いの方は、comet_IIコマンドに-vオプションを付けてください。

```
; D = 3×(A+B)-C
;
S2      START
        LD      GR0,A       GR0 ← A
        ADDA    GR0,B       GR0 ← GR0 + B ( = A+B )
        ST      GR0,D       GR0 → D
        ADDA    GR0,D       GR0 ← GR0 + D ( = D+D = 2×D = 2×(A+B) )
        ADDA    GR0,D       GR0 ← GR0 + D ( = 3×D = 3×(A+B) )
        SUBA    GR0,C       GR0 ← GR0 - C ( = 3×(A+B)-C )
        ST      GR0,D       GR0 → D ( D ← 3×(A+B)-C )
        RET
A       DC      77
B       DC      23
C       DC      30
D       DS       1
        END
```

上記のプログラムでは、GR0にA+Bを求めて、それをいったんDに格納しています。そして、GR0とDを加え、さらにもう一度Dを加えて、D+D+D=3×Dを計算し、それをGR0の新しい値としています。そして、GR0からCを引くと、3×(A+B)−Cが求められます。この値を再び、Dにストア(ST)すれば、Dの値は3×(A+B)−Cとなります。Dには、一時的に計算途中の結果(A+B)が格納されますが、最終的には3×(A+B)−Cの値がDに上書きされ、その値になることを確認して下さい。

## 演習問題5-1-2

E←2(A+B)−3(C+D)を計算するプログラムを作ってください

## 5.2 データの2倍、4倍、8倍、…、1／2倍

前節では、3倍の計算を行うのに3回加算を行いましたが、$2^n$倍の乗算あるいは$2^n$倍で割る除算を行うには、シフト演算という便利な機能があります。これは、2進数の性質を利用して、2倍、4倍、8倍というように倍々していくか、あるいは、1/2, 1/4, 1/8というように2で割った値を得る演算です。

### 5.2.1 シフト演算

**シフト演算**(shift operation)について説明しましょう。$0001_{(2)}$という2進数（10進数の1）があるとします。この各ビットを左に一つずらします。すると、$0010_{(2)}$になります。これは10進数で言えば、2であり、もとの数を2倍したものとなります。さらに各ビットを一つ左にずらします。すると、$0100_{(2)}$になり、これは10進数で言えば、4であり、もとの数を4倍したものです。つまり、2進数の各ビットを一つ左にずらすという操作はそのデータを2倍することに他なりません。このようにビットをずらす操作をシフトと呼びます。なお、シフトの結果、2進数の16桁以上になったビットは捨てられます(＊1)。

```
   1 0 1 0 1 1 0 0 0 1  =  689
  ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙
 1 0 1 0 1 1 0 0 0 1 0  = 1378  = 689×2
```

逆に右にずらすということは2で割ることに他なりません。1ビット右にシフトしたとき、最も右側のビット(ビット番号0)が1のときは、それは捨てられます。

```
 1 0 1 0 1 1 0 0 0 1 =  689
  ↘ ↘ ↘ ↘ ↘ ↘ ↘ ↘ ↘ ↘ 捨てられる
   1 0 1 0 1 1 0 0 0 =  344   (689/2 = 344.5)
```

このように、ビットを2つ左にずらす操作は4倍することになり、3つ左にずらす操作は8倍することになります。逆に、ビットを2つ右にずらす操作は4で割ることになり、3つ右にずらす操作は8で割ることになります。それでは、この規則を使ったプログラムを次項で紹介しましょう。

**ポイント**

ビットを左に1つシフトすると2倍になり、ビットを右に1つシフトすると1/2になる

**演習問題5-2-1**

ビットを左に1つシフトすると2倍になり、ビットを右に1つシフトすると1/2になるという理由を2進数の定義の式を用いて説明しなさい。

(＊1) 実際には、符号ビットの扱いなどがあり、もう少し複雑な処理になりますが、それについては、6.5節で説明します。

### 5.2.2 データを8倍するプログラム

それでは、データを8倍するプログラムを紹介しましょう。

```
;          シフト命令でデータを8倍する
MUL8       START
           LD        GR1,X
           SLA       GR1,3     GR1 ← GR1 × 8
           ST        GR1,Y
           RET
X          DC        15
Y          DS        1
           END
```

ビットを左にシフトするシフト命令は、SLA命令を用います。「SLA　汎用レジスタ,シフト数」という形式でビットを指定された数だけ左にシフトすることができます。上記の例では、「SLA　GR1,3」ですから、GR1の値を左に3ビットシフトします。これは、レジスタGR1の値を$2^3$＝8倍することになります。このプログラムを実行すると、15×8＝120がYに格納されます。

このプログラムを1000(16)番地にロードして実行した結果を示します。

```
 PR  Instruction     SP   OSZ  GR0  GR1  GR2  GR3  GR4  GR5  GR6  GR7  Explanation
1000 LD    GR1,1007 7530 --- 8029 000F 98BE E784 CAE1 BD6C ACD6 F2AE 1007:[h:000F, s:15, u:15]
1002 SLA   GR1,0003 7530 --- 8029 0078 98BE E784 CAE1 BD6C ACD6 F2AE left 3 shift => GR1:[s:120]
1004 ST    GR1,1008 7530 --- 8029 0078 98BE E784 CAE1 BD6C ACD6 F2AE GR1:[h:0078, s:120, u:120]
1006 RET            7530 --- 8029 0078 98BE E784 CAE1 BD6C ACD6 F2AE Return to OS
```

最終的にGR1には120(＝78(16))が入ってそれが領域Y(1008番地)に格納されたことが分かるでしょう。

## 演習問題5-2-2

(1) シフト命令を用いてデータを4倍するプログラム(EX522A)を作りなさい。

(2) シフト命令を用いてデータを8倍するプログラム(EX522B)を作りなさい。

(3) シフト命令を用いてA＋Bを8倍するプログラム(EX522C)を作りなさい。

### 5.2.3 10×(A＋B＋C)を計算するプログラム

次に、10×(A＋B＋C)を計算するプログラムを紹介します。

```
;         D ← 10 × ( A+B+C )
MUL10     START
          LD        GR1,A
          ADDA      GR1,B
          ADDA      GR1,C     GR1 ← A+B+C
          ST        GR1,D     D ← A+B+C
          SLA       GR1,3     GR1 ← 8 × ( A+B+C )
          ADDA      GR1,D     GR1 ← 9 × ( A+B+C )
          ADDA      GR1,D     GR1 ← 10 × ( A+B+C )
          ST        GR1,D     D ← 10 × ( A+B+C )
          RET
A         DC        1
B         DC        2
C         DC        3
D         DS        1
          END
```

これは、コメントにも示したとおり、まず、A+B+Cの値をGR1に求めて、それをシフト命令で8倍にした後、D(=A+B+C)を2回加えて、10倍しています。

しかし、次のアルゴリズムによるプログラムも考えられます。

```
;         D ← 10 × ( A+B+C )
MUL10A    START
          LD        GR1,A
          ADDA      GR1,B
          ADDA      GR1,C     GR1 ← A+B+C
          SLA       GR1,1     GR1 ← 2 × ( A+B+C )
          ST        GR1,D     D ← 2 × ( A+B+C )
          SLA       GR1,2     GR1 ← 4 × { 2 × ( A+B+C ) } = 8 × ( A+B+C )
          ADDA      GR1,D     GR1 ← {8(A+B+C)+2(A+B+C)} =10 × ( A+B+C )
          ST        GR1,D     D ← 10 × ( A+B+C )
          RET
A         DC        1
B         DC        2
C         DC        3
D         DS        1
          END
```

これは、まず、GR1に2 × ( A+B+C )の値を計算し、それをDに退避させます。そして、GR1(=2×( A+B+C ))を4倍(すなわち、8×( A+B+C ))した値を求めて、そこにD(2×( A+B+C ))の値を加えて、10 × (A+B+C )を計算しています。

しかし、このプログラム例では、途中の計算結果の保存を主記憶装置に対して行っています。実際には、アクセス効率の良い汎用レジスタに退避する方が良いでしょう。COMET IIには8個の汎用レジスタがあり、極力、これを有効に使う方が良いでしょう(この例については、演習問題5-2-3参照)。

## 演習問題5-2-3

途中の計算結果の保存を汎用レジスタに対して行うようにして次のプログラムを作成しなさい。なお、汎用レジスタの値を別の汎用レジスタに複写するには、たとえば「LD GR1,GR2」とします。これでGR2の値がGR1に複写されます。

(1) データを11倍するプログラム(EX523A)を作りなさい。

(2) A+Bを11倍するプログラム(EX523B)を作りなさい。

### 5.2.4 2つの数の平均を計算するプログラム

AとBとの平均を計算するプログラムを示します。このプログラムでは、A+Bの値をビットを1つ右にシフトすることによって、2で割って値を求めています。つまり、(A+B)/2、すなわちAとBとの平均を求めていることになります。もちろん、整数の割り算ですから、小数点以下は切り捨てられることになります。ビットを右にシフトするには、SRA命令を用います。

```
;        AとBの平均を計算する
MEAN1    START
         LD      GR1,A
         ADDA    GR1,B
         SRA     GR1,1    GR1 ← ( A+B ) / 2
         ST      GR1,C    C ← ( A+B ) / 2
         RET
A        DC      10
B        DC      20
C        DS      1
         END
```

1000番地からロードして実行した結果を次に示します(*1)。

```
 PR  Instruction       SP   OSZ  GR0  GR1  Explanation
1000 LD     GR1,1009  7530  ---  8029 000A 1009:[h:000A, s:10, u:10]
1002 ADDA   GR1,100A  7530  ---  8029 001E 100A:[h:0014, s:20] +=> GR1:[s:30]
1004 SRA    GR1,0001  7530  ---  8029 000F right 1 shift => GR1:[s:15]
1006 ST     GR1,100B  7530  ---  8029 000F GR1:[h:000F, s:15, u:15]
1008 RET              7530  ---  8029 000F Return to OS
```

ラベルCの領域(100B番地)に10+20=30を2で割った値 15 が格納されたことが分かります。

## 演習問題5-2-4

(1) データを8で割るプログラム(EX524A)を作りなさい。

(2) A+Bを16で割るプログラム(EX524B)を作りなさい。

(*1) 紙面の都合上、このプログラムでは使用していないGR2～GR7の値は表示しないことにします。

# 5.3 指標レジスタの利用

本節では、アセンブラ言語を習得する上で、どうしても理解しなければならない、**指標レジスタ**(index register)について説明します。COMET IIでは、汎用レジスタGR1～GR7を指標レジスタという特殊な用途に用いることができます。指標レジスタを使うと、さまざまな処理を行うことができます。たとえば、「先頭から3番目のデータ」、「先頭から10番目のデータ」という処理ができます。この機能によってアセンブラプログラミングの幅を大きく広げています。

## 5.3.1 指標レジスタの使用例：例題を通して

次の例題を用いて、指標レジスタを説明していきましょう。

8名の学生に対して試験を行ったところ、次の点数を取ったとします。

**表5-1 試験結果**

| 学生番号 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 点　　数 | 56 | 87 | 89 | 77 | 90 | 34 | 76 | 64 |

学生番号が0から始まっていますが、0から始まるようにしておいた方が計算がしやすいからです。その理由は、このプログラムの動作の仕組みを理解すれば、分かります。

さて、領域STIDには、学生番号が格納されているとしましょう。STIDに格納された学生番号を持つ学生の点数を、領域POINTに格納するプログラムを作ることを考えてみましょう。たとえば、STIDに5が入っていれば、学生番号5の点数34がPOINTに格納されるようにしたいのです。

それでは、このプログラムについて考えてみましょう。まず、上記の表5-1のデータを格納しなければなりません。これは、次に示すように、DC命令を用いて行います。

```
STUDENT    DC    56
           DC    87
           DC    89
           DC    77
           DC    90
           DC    34
           DC    76
           DC    64
```

今までのDC命令とは異なり、最初のDC命令だけにラベルSTUDENTが付いています。指標レジスタを使う場合、先頭のアドレスだけが分かればよいので、このようになっています。アセンブラ言語では、このように同じ種類の連続するデータを一般に**表**(table)と呼び、本書では第11章で詳しく説明します。

なお、一つのDC命令にはデータをコンマで区切っていくつでも書けますから、次のようにすることもできます。

```
STUDENT    DC    56,87,89,77,90,34,76,64
```

この2つの書き方の異なるプログラム(DC命令)は、ともにまったく同じ機械語に変換されます。

今回は、各点数の横にコメントを書きたいので、以下のプログラムのようにしました。

```
INDEX1     START
           LD    GR1,STID
           LD    GR0,STUDENT,GR1
           ST    GR0,POINT
           RET
STUDENT    DC    56     学生番号(STID=0)の人の点数
           DC    87     学生番号(STID=1)の人の点数
           DC    89     学生番号(STID=2)の人の点数
           DC    77     学生番号(STID=3)の人の点数
           DC    90     学生番号(STID=4)の人の点数
           DC    34     学生番号(STID=5)の人の点数
           DC    76     学生番号(STID=6)の人の点数
           DC    64     学生番号(STID=7)の人の点数
STID       DC     5     表示する学生の番号
POINT      DS     1     STIDの人の点数を格納するところ
           END
```

これをアセンブラにかけて機械語を生成したときのリストを示しましょう。また、説明の都合上、このプログラムは0番地からロードされて実行されるとします。

```
  Machine    Source List (temp.casl)
Adr. Code Line                Program
    |    |  1| INDEX1     START
   0|1010|  2|            LD    GR1,STID
   1|000F|
   2|1001|  3|            LD    GR0,STUDENT,GR1
   3|0007|
   4|1100|  4|            ST    GR0,POINT
   5|0010|
   6|8100|  5|            RET
   7|0038|  6| STUDENT    DC    56     学生番号(STID=0)の人の点数
   8|0057|  7|            DC    87     学生番号(STID=1)の人の点数
   9|0059|  8|            DC    89     学生番号(STID=2)の人の点数
   A|004D|  9|            DC    77     学生番号(STID=3)の人の点数
   B|005A| 10|            DC    90     学生番号(STID=4)の人の点数
   C|0022| 11|            DC    34     学生番号(STID=5)の人の点数
   D|004C| 12|            DC    76     学生番号(STID=6)の人の点数
   E|0040| 13|            DC    64     学生番号(STID=7)の人の点数
   F|0005| 14| STID       DC    5      表示する学生の番号
  10|8029| 15| POINT      DS    1      STIDの人の点数を格納するところ
    |    | 16|            END
```

ラベルSTUDENTのアドレスが、7番地(網かけ)になっていることを確認して下さい。それでは、このプログラムの動作を見ていきましょう。「LD　GR1,STID」によって、STIDの値(5)がGR1にロードされます。

それでは、次の命令はどのような意味なのでしょうか。

```
LD      GR0,STUDENT,GR1
                     ↑
                 指標レジスタ
```

これは、今まで見てきた命令とは、書き方が多少異なります。最後に記述されているレジスタGR1が指標レジスタです。この命令では、ラベルSTUDENTのアドレス(7番地)とGR1の値(5)の合計(12番地＝C(16)番地)を求め、その合計値のアドレス(12番地＝C(16)番地)の内容(34＝0022(16))をGR0にロードします。この例では、次のようになります。

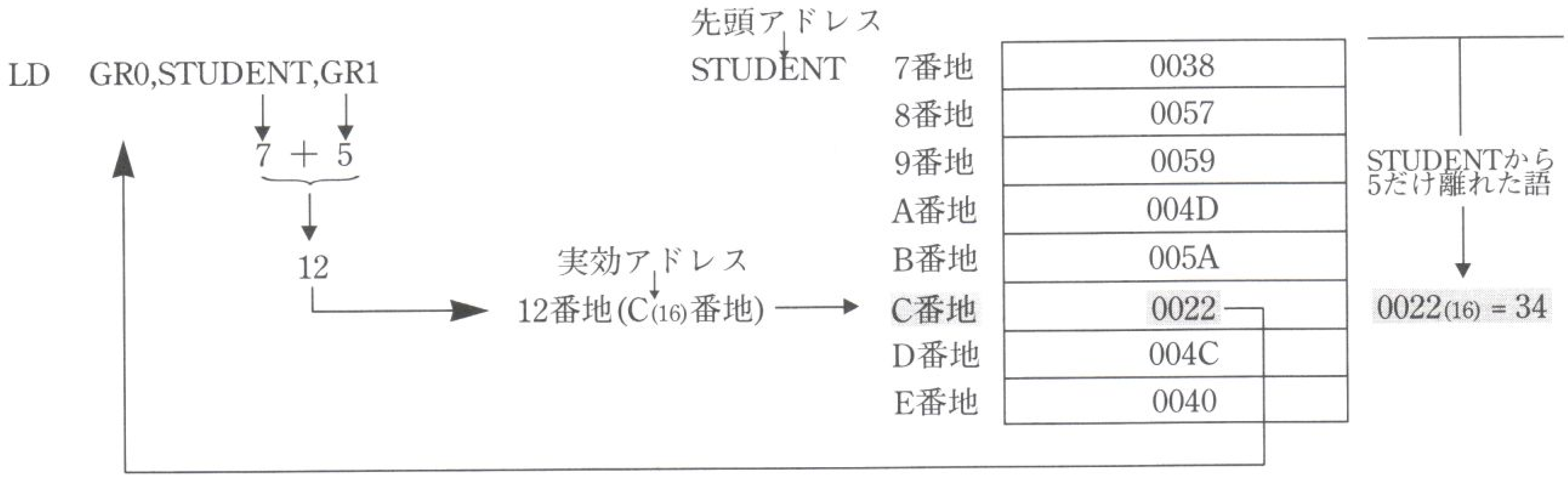



**図5-4 「LD GR0,STUDENT,GR1」の意味**

つまり、この命令は先頭(STUDENT)からGR1(5)だけ離れた場所(ただし、先頭(7番地)を「0だけ離れた場所」、8番地を「1だけ先頭から離れた場所」と考えます)の値をGR0にロードすることになります。

このように指標レジスタを使うと、何番目の値(この例では5番目の値)という指示ができるのでとても便利です。指標レジスタを使って生成されたアドレス(上記の例では12番地)を**実効アドレス**(effective address)と呼びます。次のようにGR1の値が変われば、実効アドレスも変わります。




**図5-5 GR1の値によって実効アドレスが変わる**

ところで、指標レジスタとして使えるのは、GR1～GR7までで、GR0は使えませんので注意してください。

それでは、このプログラムの実際の動作を見ていきましょう。今回も生成されるアドレスを分かりやすく見るために、前述のように0番地から機械語のプログラムをロードしましょう。

```
 PR  Instruction        SP   OSZ  GR0  GR1  Explanation
0000 LD   GR1,000F      7530 ---  8029 0005 000F:[h:0005, s:5, u:5]
0002 LD   GR0,0007,GR1  7530 ---  0022 0005 000C:[h:0022, s:34, u:34]
0004 ST   GR0,0010      7530 ---  0022 0005 GR0:[h:0022, s:34, u:34]
0006 RET                7530 ---  0022 0005 Return to OS
```

最初の命令で、F$_{(16)}$番地(すなわち、ラベルSTIDの領域)の値(5)がレジスタGR1にセットされています。次の命令「LD GR0,0007,GR1」には、指標レジスタGR1が付いています。このGR1の中には5がセットされていますから、その値にアドレス0007が加算されたアドレス(12番地、16進数ではC$_{(16)}$番地)が実効アドレスとなって、その番地の値(34)がGR0にロードされます。なお、上記のトレース結果の右側の網掛け部分(000F, 000C)は、その命令によって生成された実効アドレスを示しています。

また、上記のトレース結果に示されている「LD GR0,0007,GR1」は、アセンブラ言語のプログラム「LD GR0,STUDENT,GR1」をアセンブルした機械語を主記憶装置にロードしたものを再びニーモニックに直したものです。したがって、ラベルSTUDENTがロード先の絶対アドレスになっている点に注意してください。

このように機械語から元のニーモニックコードに戻すことを逆アセンブルすると呼び、それを行うソフトウェアを**逆アセンブラ**と呼びます。

## 演習問題5-3-1

なぜ、表の先頭の語を「0番め」とするのでしょうか。

### 5.3.2 さまざまな命令で使われる指標レジスタ

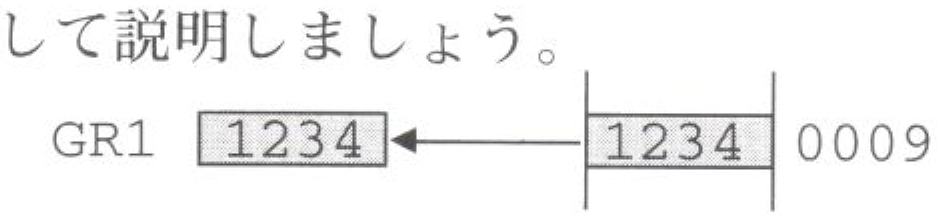
指標レジスタは、LD命令だけでなく、ST命令やADDA命令など、多くの機械語命令で使えます。そのため、指標レジスタが使える命令の構文は、次のように記述されます。

```
LD   r,adr[,x]
```

ここで、rは汎用レジスタ(GR0～GR7)、adrはアドレス(ラベルあるいは10進定数か16進定数で番地を直接指定)を意味しています。次の[ ]の中身は省略できることを意味します。その中に ,x とありますが、これは、その部分を省略しない場合には、,（コンマ）のあとにx(指標レジスタ:GR1～GR7)を指定する事を意味しています。なお、指標レジスタが省略された場合、実効アドレスはadrだけで決定されます。

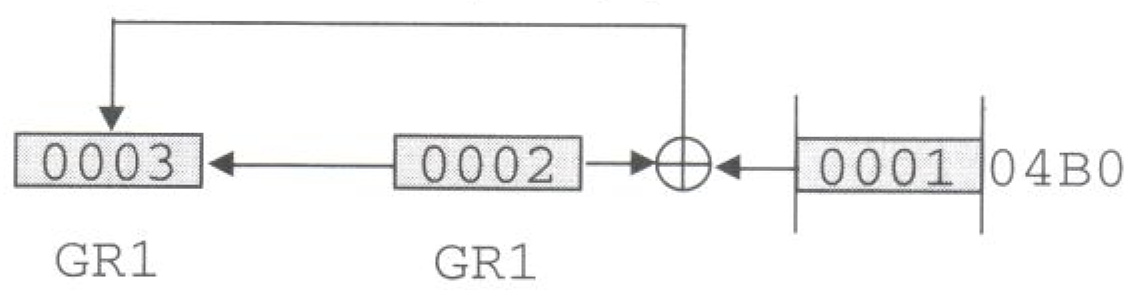
なお、実効アドレスについて例を示して説明しましょう。

```
LD    GR1,#0009
```



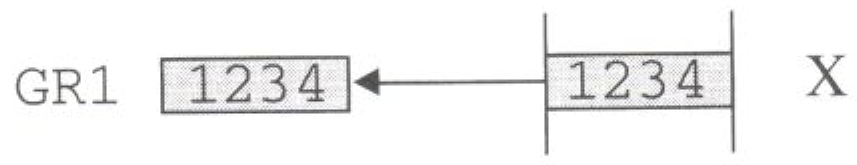

上記の場合、指標レジスタがないので、9$_{(16)}$番地がそのまま実効アドレスとなります。したがって、実効アドレス(9$_{(16)}$番地)の内容1234$_{(16)}$がGR1にロードされます。

```
ADDA GR1,1200
```



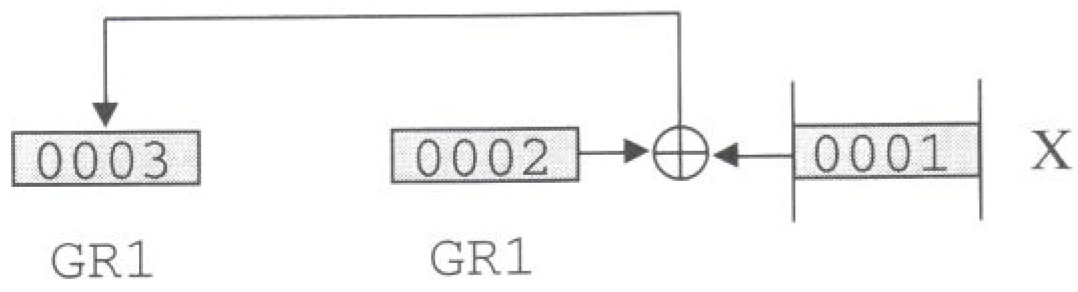

上記の例でも、指標レジスタがないので、1200$_{(10)}$（＝04B0$_{(16)}$）番地が実効アドレスとなります。実効アドレスの値1が、レジスタGR1に加算されます。

```
LD    GR1,X
```



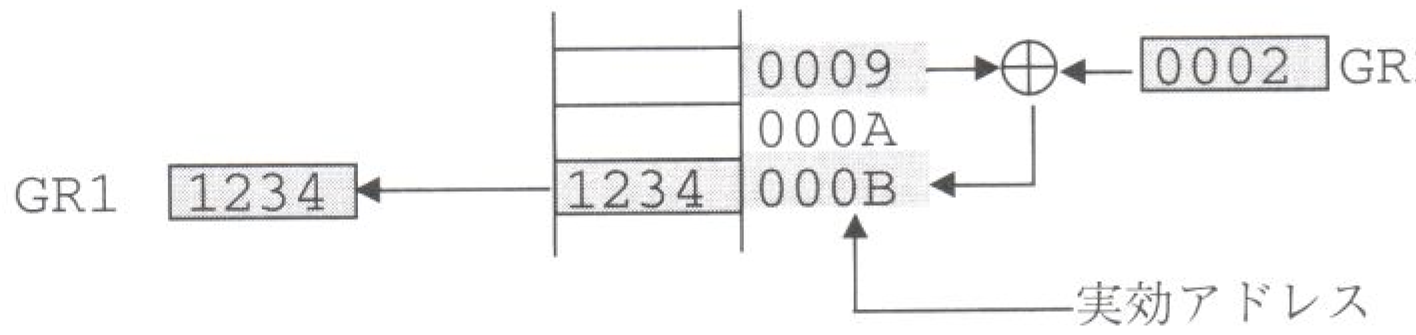

上記の例では、ラベルXのアドレスが実効アドレスとなります。そのアドレスの内容$1234_{(16)}$がGR1にロードされます。

```
ADDA GR1,X
```



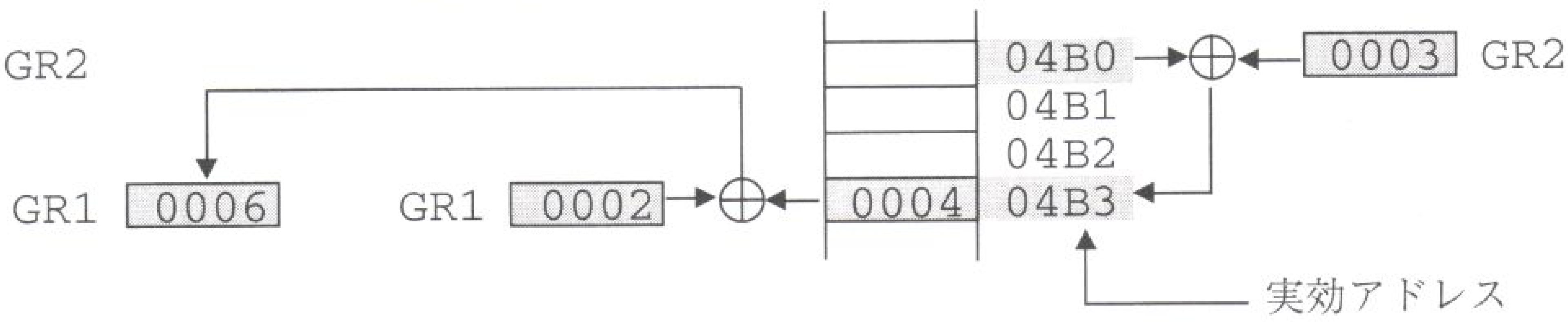

上記の例では、指標レジスタがないので、ラベルXのアドレスが実効アドレスとなります。したがって、ラベルXの内容1がGR1に加算されます。

```
LD    GR1,#0009,GR2
```



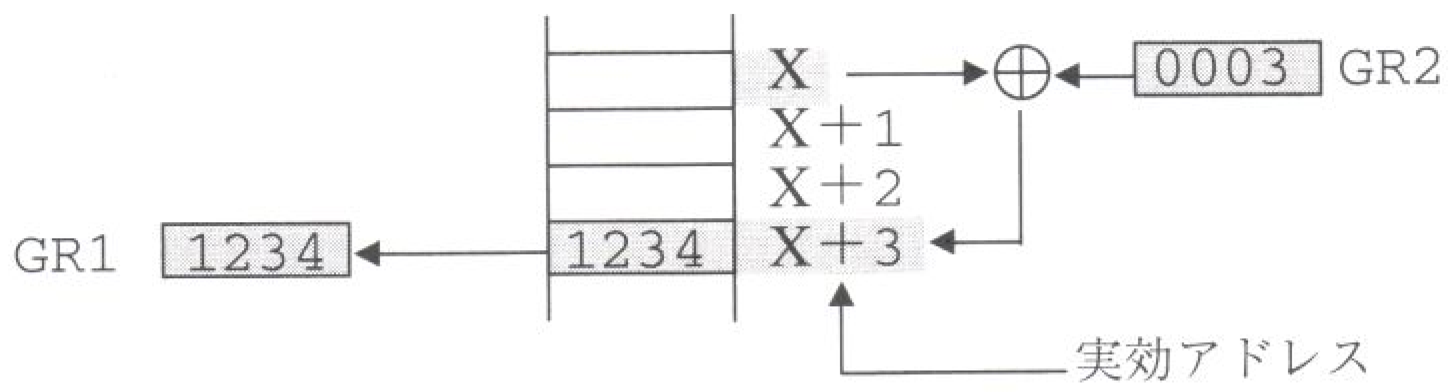

上記の場合、指標レジスタGR2がありますので、「$9_{(16)}$番地＋GR2の内容」である$000B_{(16)}$が実効アドレスとなります。実効アドレスの内容$1234_{(16)}$がGR1にロードされます。

```
ADDA GR1,1200,GR2
```



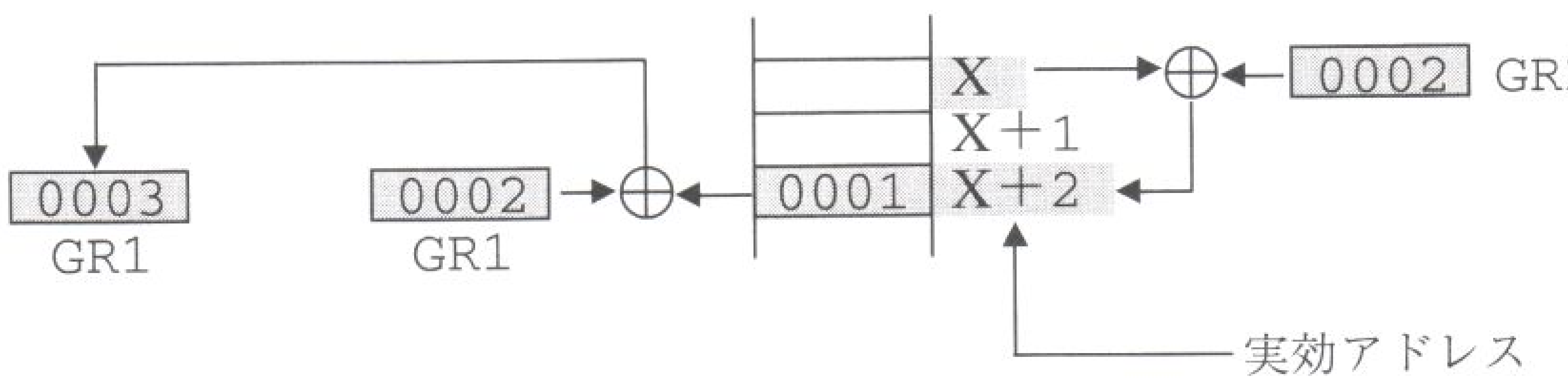

上記の例では、指標レジスタGR2がありますので、「$1200_{(10)}$＝$4B0_{(16)}$番地＋GR2の内容」である$04B3_{(16)}$が実効アドレスとなります。そして、実効アドレスの値4が、レジスタGR1に加算されます。

```
LD    GR1,X,GR2
```




上記の例では指標レジスタGR2がありますので、「ラベルXのアドレス＋GR2の内容」が実効アドレスとなります。そのアドレスの内容$1234_{(16)}$がGR1にロードされます。

```
ADDA GR1,X,GR2
```




上記の例では、指標レジスタGR2がありますので、「ラベルXのアドレス＋GR2の内容」が実効アドレスとなります。そして、実効アドレスの内容1がGR1に加算されます。

## 演習問題5-3-2

ラベルTABLE1とTABLE2は連続して記述され、次のデータが定義されているとします。

```
TABLE1  DC  100,101,102,103,104,105
TABLE2  DC  200,201,202,203,204,205
```

このとき、次の各命令でレジスタGR0に格納される値はいくつですか。

(1) LD GR0,TABLE1

(2) LD GR0,TABLE2

(3) LD GR0,TABLE1,GR1 (GR1には2が格納されている)

(4) LD GR0,TABLE1,GR2 (GR2には7が格納されている)

(5) LD GR0,TABLE2,GR3 (GR3には4が格納されている)

(6) LD GR0,TABLE2,GR4 (GR4には−1が格納されている)

**♥CASLDVにおけるさまざまな表示機能♥**

CASLDVには、読者がCASL IIのプログラミングを効率よく学習できるように、さまざま表示機能が用意されています。それらの表示機能を使うには、メニューの中から「表示」を選択します。

「OUT命令の出力表示」を選択すると、OUT命令によって出力されるデータを表示するダイアログが表示されます。OUT命令による出力は、インフォメーションペインにも出力されますが、他のトレース情報の中に埋没することが多いので、このダイアログが用意されています。

「レジスタの表示」は、汎用レジスタの値を、さまざまな表現方法で表示します。ここに表示されているレジスタの値は、プログラムの実行と共にリアルタイムに変化します。

「記憶領域の表示」は、主記憶装置の中を表示する機能ですが、ラベルで指定したり、直接アドレスで指定することができます。

「メモリマップとロード状況」は、COMET IIの主記憶装置の概観を示し、プログラムのロード場所やスタックの位置などを表示します。

## 5.4 フラグレジスタと条件判断

今までのプログラムは、上から順番に実行されるストレートなものでした。しかし、これでは、大したプログラムは作成できません。プログラムが複雑な処理を実現できるのは、条件判断とループ(繰り返し)の機能を持っているからなのです。ここでは、条件によってプログラムの動作を変更する条件判断のプログラムについて説明しましょう。

たとえば、領域Xの値と領域Yの値の大きい方を領域Zに格納するプログラムについて考えてみましょう。このプログラムの大まかな流れは、次のようになります。

```
(1) GR1 ← X
(2) GR2 ← Y
(3) GR1とGR2を比較し、GR1<GR2なら (6)へ移る
(4) GR3 ← GR1 (GR1の方が大きいから)
(5) (7)へ移る
(6) GR3 ← GR2 (GR2の方が大きいから)
(7) 終了する
```

このプログラムは、(1)→(2)→(3)と実行され、(3)で、GR1<GR2であれば、プログラムの次の実行を(4)ではなく、(6)にしています。プログラムの実行をその次の命令ではなく、他の命令に移すことを**分岐**あるいは**ジャンプ**と呼びます。この流れをプログラム化すると、次のようになります。

```
BIG     START
        LD      GR1,X       GR1 ← X
        LD      GR2,Y       GR2 ← Y
        CPA     GR1,GR2     GR1とGR2を比較する
        JMI     SMALL       もし、GR1<GR2なら SMALL へジャンプ
        ST      GR1,Z       GR1 → Z, GR1の方が大きい
        JUMP    FINAL       FINALへジャンプ
SMALL   ST      GR2,Z       GR2 → Z, GR2の方が大きい
FINAL   RET
X       DC      5
Y       DC      8
Z       DS      1
        END
```

さて、ここで新しい命令が出てきました。「CPA GR1, GR2」という機械語命令は、レジスタGR1の値とレジスタGR2の値を比較するという命令です。この命令の実行により、フラグレジスタの値が変わります。フラグレジスタには、OF、SF、ZFという3つのビットがあります。「CPA GR1, GR2」命令によって、SFおよびZFの内容が次の表に従って設定されます(*1)。

(*1)「CPA」命令では、OFの内容は常に0になります。

**表 5-2 フラグレジスタの変化**

| フラグ | 説　明 |
|---|---|
| SF | GR1<GR2のとき1、GR1≧GR2のとき0 |
| ZF | GR1=GR2のとき1、GR1≠GR2のとき0 |

このプログラムの例では、GR1の値が5、GR2の値が8ですから、GR1<GR2が成立し、「CPA　GR1,GR2」によってフラグレジスタの値は次のようになります。

**表5-3 今回のプログラム「CPA　GR1,GR2」の結果**

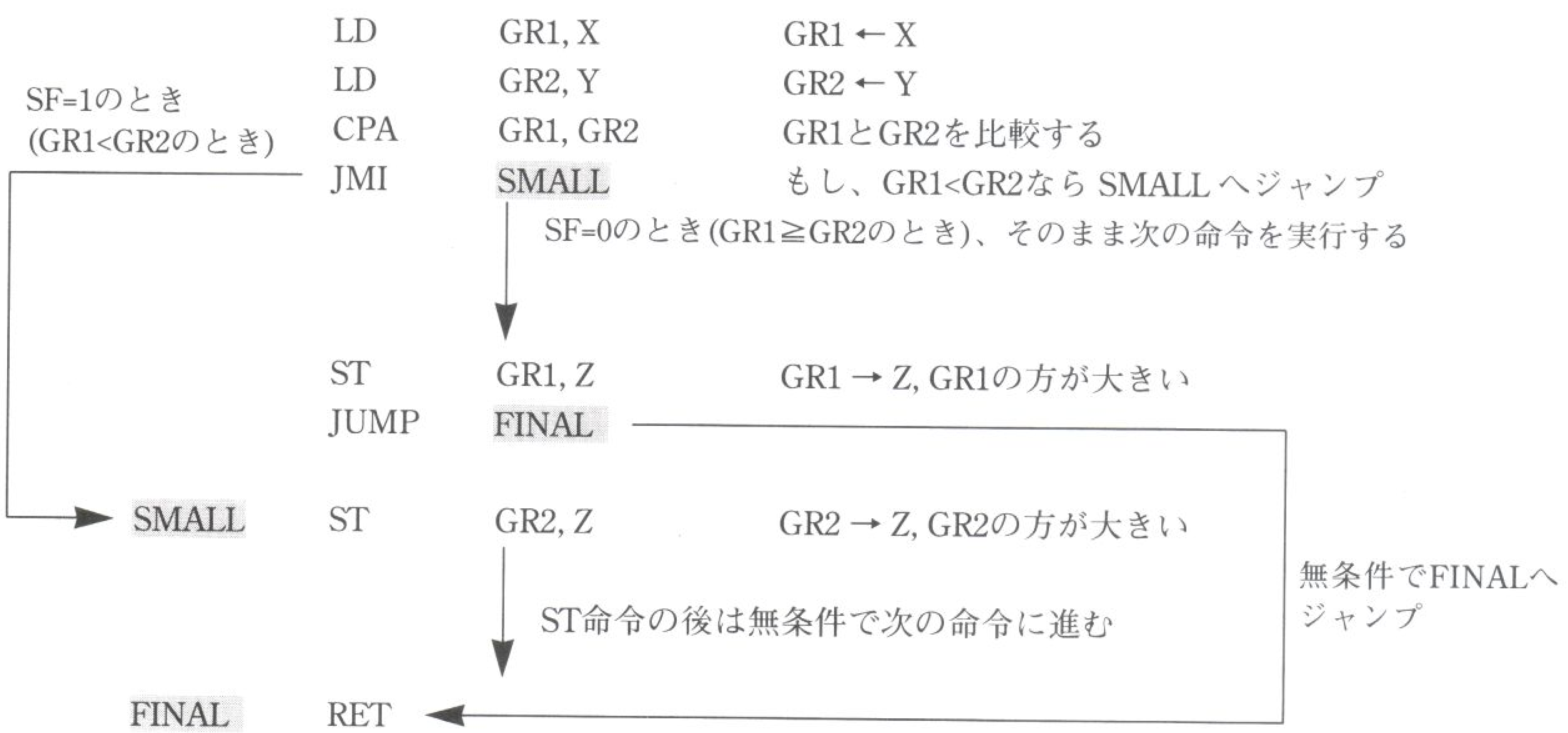
| フラグ | 値 |
|---|---|
| OF | 0 |
| SF | 1 |
| ZF | 0 |

さて、CPA命令の次にある「JMI　SMALL 」という命令は、このフラグレジスタのSFビットを見て、そのビットが立っていたら[*1]、プログラムの次の実行をラベルSMALLの位置の機械語命令からに移すという命令です。

プログラムの動作は、通常は上から下に進むのですが、この命令は、その実行の順番を変更する命令です。動作の実行を変えることを前述のように「ジャンプする」といいます。この命令の場合、「SFビットが立っていれば、SMALLにジャンプする」ということになります。もし、SFビットが立っていなければ、プログラムの実行は「ジャンプする」ことなく、次の命令「ST　GR1, Z」が実行されます。

ところで、次のプログラムにはこの命令の次にも新しい命令があります。それは、「JUMP FINAL」命令ですが、この命令は「無条件で(常に)実行をFINALにジャンプ」させます。

したがって、このプログラムの流れを図示すると、次のようになります。




(＊1)　ビットの値が1のとき「ビットが立つ」といいます。

このプログラムの場合、GR1(5)<GR2(8)ですから、SF=1となります。したがって、「JMI　SMALL」でジャンプし、「ST GR2,Z」が実行され、GR2(8)がZに格納されます。つまり、Zには、大きい方の値8が格納されます。

それでは、実際にこのプログラムを動作させてみることにしましょう。アドレスを確認するために、アセンブリリストを示します。

```
  Machine    Source List
Adr.  Code Line           Program
          |    1 | BIG        START
    0 | 1010|  2 |            LD    GR1,X      GR1 ← X
    1 | 000E|    |
    2 | 1020|  3 |            LD    GR2,Y      GR2 ← Y
    3 | 000F|    |
    4 | 4412|  4 |            CPA   GR1,GR2    GR1とGR2を比較する
    5 | 6100|  5 |            JMI   SMALL      もし、GR1<GR2なら SMALL へジャンプ
    6 | 000B|    |
    7 | 1110|  6 |            ST    GR1,Z      GR1 → Z, GR1の方が大きい
    8 | 0010|    |
    9 | 6400|  7 |            JUMP  FINAL      FINALへジャンプ
    A | 000D|    |
    B | 1120|  8 | SMALL      ST    GR2,Z      GR2 → Z, GR2の方が大きい
    C | 0010|    |
    D | 8100|  9 | FINAL      RET
    E | 0005| 10 | X          DC    5
    F | 0008| 11 | Y          DC    8
   10 | 8029| 12 | Z          DS    1
          |   13 |            END
```

SMALLラベルの位置はB(16)番地、FINALラベルの位置はD(16)番地ですね。この番地が、実行の番地と一致するようにするために、0番地からロードすることにしましょう。実行結果は次のようになりました。以下の部分で、OSZと書いてある部分がフラグレジスタの内容です。ビットが立っていない状態(0のとき)を ー で示し、立っている状態をそれぞれのビット名の名前(SFならS)で示します。

```
 PR  Instruction     SP   OSZ  GR0  GR1  GR2   Explanation
0000 LD   GR1,000E   7530 ---  8029 0005 98BE  000E:[h:0005, s:5, u:5]
0002 LD   GR2,000F   7530 ---  8029 0005 0008  000F:[h:0008, s:8, u:8]
0004 CPA  GR1,GR2    7530 -S-  8029 0005 0008  GR1:[s:5] ? GR2:[s:8]
0005 JMI  000B       7530 -S-  8029 0005 0008  jump to 000B
000B ST   GR2,0010   7530 -S-  8029 0005 0008  GR2:[h:0008, s:8, u:8]=> 0010
000D RET             7530 -S-  8029 0005 0008  Return to OS
```

「CPA　GR1, GR2」を実行すると、GR1<GR2なのでSF=1(Sと表示されている)となっていることが分かります。この結果、次の「JMI　000B」を実行すると、実際に000B(16)番地に実行が移っている(PRが000Bになっている)ことがわかります。そして、「ST GR2,0010」を実行し、ラベルZの領域に8が格納されます。

このプログラムでは、7番地目の「ST GR1,Z」と9番地目の「JUMP FINAL」はまったく実行されていません。条件判断を含むプログラムでは，この例のようにまったく実行されない部分が出てきます。

さて、それでは、領域Xの値を15に変えて、もう一度実行してみましょう。

```
X       DC      15
Y       DC      8
Z       DS      1
```

このプログラムを同じように0番地からロードして実行すると、次のようになりました。

```
 PR  Instruction     SP   OSZ  GR0  GR1  GR2   Explanation
0000 LD   GR1,000E  7530  ---  8029 000F 98BE  000E:[h:000F, s:15, u:15]
0002 LD   GR2,000F  7530  ---  8029 000F 0008  000F:[h:0008, s:8, u:8]
0004 CPA  GR1,GR2   7530  ---  8029 000F 0008  GR1:[s:15] ? GR2:[s:8]
0005 JMI  000B      7530  ---  8029 000F 0008  no jump
0007 ST   GR1,0010  7530  ---  8029 000F 0008  GR1:[h:000F, s:15, u:15]=> 0010
0009 JUMP 000D      7530  ---  8029 000F 0008  jump to 000D
000D RET            7530  ---  8029 000F 0008  Return to OS
```

今度は、GR1(15)＞GR2(8)ですので、SFが立ちません。したがって、「JMI 000B」を実行してもジャンプしません。プログラムの実行は、次の7番地の「ST　GR1,0010」になります。この命令で、GR1の値(15)をラベルZに格納します。その後、9番地の「JUMP　000D」を実行し、無条件で$D_{(16)}$番地にジャンプし、$D_{(16)}$番地のRETを実行し、プログラムの実行が終了します。したがって、この場合、B番地の「ST　GR2, Z」が実行されません。

## 演習問題5-4

次のプログラムで命令「LD　GR1,X」は実行されるでしょうか。

```
        JUMP  SKIP
        LD    GR1,X
   SKIP LD    GR2,Y
```

# 5.5 リテラル

レジスタGR0に7という値を入れたいとします。今までの知識では、次のようにプログラムを書く必要がありました。

```
        LD      GR0,SEVEN  ←———— 引用場所
        ↓
SEVEN   DC      7          ←———— 定義場所
```

つまり、DC命令を使って、主記憶装置のどこかに7を格納した語を配置し、それにラベルを付けて、そこを参照する方法です。しかし、この方法では、引用場所と定義場所が別々の場所にあるので、間違えることもしばしばあります。

そこで、CASL IIには、**リテラル**という便利な記述方法があります。これは、次の左のように書くと、これを右のような機械語に変換して出力するというものです。

```
LD      GR0,=7
```

→

```
        LD      GR0,temp
        ↓
temp    DC      7
```

tempという名前は、ここで説明するために仮想的に付けた名前で実際にtempというラベルが作られるわけではありません。つまり、「=7」という記述があると、CASL IIアセンブラが7というデータを格納した語を用意し、「=7」の部分にはそこへのアドレスを格納するのです。これにより、その値が対応するレジスタへロードされるわけです。

もともと、COMET IIにはリテラルを扱う命令はありません。CASL IIアセンブラがソースプログラムをCOMET IIの機械語に翻訳するときにこの変換処理を行ってくれるのです。リテラルを用いると、汎用レジスタへのデータのセット（複写）を直感的に示すことができて大変便利です。

さて、CASL IIの文法的側面から、もう少しリテラルについて説明しましょう。リテラルは、機械語命令のアドレスを書く部分(一般形式でadrと示している部分)へ次の形式のものを指定する構文です。

**表5-4 リテラルの構文**

| 構　文 | 例 | | |
|---|---|---|---|
| = 10進定数 | =5 | =1000 | =-23 |
| = 16進定数 | =#1230 | =#FF12 | =#AAAA |
| = 文字定数 | ='a' | ='B' | ='abc' |

この方法を採用したプログラムを以下に示します。GR0には、ラベルを用いた方法で7が格納されています。GR1とGR2には、リテラルを使ってそれぞれ6とFF(16)をロードしています。また、GR1にリテラルを使って2を加算し、GR2からリテラルを使って、F0(16)を減算しています。

```
LITRL   START
        LD      GR0,SEVEN       通常の方法
        LD      GR1,＝6         GR1←6  リテラルを利用
        LD      GR2,＝#00FF     GR2←FF(16)  リテラルを利用
        LD      GR3,＝'a'       GR3←'a'  リテラルを利用
        ADDA    GR1,＝2         GR1←GR1＋2  リテラルを利用
        SUBA    GR2,＝#00F0     GR2←GR2-F0(16)  リテラルを利用
        RET
SEVEN   DC      7
        END
```

このプログラムをアセンブラにかけて、どんな機械語が生成されるか調べてみましょう。

```
   Machine      Source List (LITRL.casl)
 Adr.Code   Line   Program
   |      |    1 | LITRL    START
  0| 1000 |    2 |          LD      GR0,SEVEN     通常の方法
  1| 000D |      |
  2| 1010 |    3 |          LD      GR1,＝6       GR1←6  リテラルを利用
  3| 000E |      |
  4| 1020 |    4 |          LD      GR2,＝#00FF   GR2←FF(16)  リテラルを利用
  5| 000F |      |
  6| 1030 |    5 |          LD      GR3,＝'a'     GR3←'a'  リテラルを利用
  7| 0010 |      |
  8| 2010 |    6 |          ADDA    GR1,＝2       GR1←GR1＋2  リテラルを利用
  9| 0011 |      |
  A| 2120 |    7 |          SUBA    GR2,＝#00F0   GR2←GR2-F0(16)  リテラルを利用
  B| 0012 |      |
  C| 8100 |    8 |          RET
  D| 0007 |    9 | SEVEN    DC      7
  E| 0006 |      | (リテラルによって生成)
  F| 00FF |      | (リテラルによって生成)
 10| 0061 |      | (リテラルによって生成)
 11| 0002 |      | (リテラルによって生成)
 12| 000F |      | (リテラルによって生成)
   |      |   10 |          END
```

上記の例にあるように、リテラルで生成されるデータはEND命令の直前に置かれます。
それでは、このプログラムを実行してみましょう。

```
 PR  Instruction      SP   OSZ  GR0  GR1  GR2  GR3  Explanation
0000 LD   GR0,000D   7530  ---  0007 C823 98BE E784  000D:[h:0007, s:7, u:7]
0002 LD   GR1,000E   7530  ---  0007 0006 98BE E784  000E:[h:0006, s:6, u:6]
0004 LD   GR2,000F   7530  ---  0007 0006 00FF E784  000F:[h:00FF, s:255, u:255]
0006 LD   GR3,0010   7530  ---  0007 0006 00FF 0061  0010:[h:0061, s:97, u:97]
0008 ADDA GR1,0011   7530  ---  0007 0008 00FF 0061  0011:[h:0002, s:2] +=> GR1:[s:8]
000A SUBA GR2,0012   7530  ---  0007 0008 000F 0061  0012:[h:000F, s:15] -=> GR2:[s:240]
000C RET             7530  ---  0007 0008 000F 0061  Return to OS
```

## 演習問題5-5

(1) 次のプログラムをリテラルで記述してください。

(a) レジスタGR0に'a'を格納する。

(b) レジスタGR1に−45を格納する。

(c) レジスタGR2に−1を16進表記で格納する。

(2) 次のプログラムの実行でレジスタGR2には何が格納されるでしょうか。

```
LITEST START
       LD GR2,='CASL II'
       RET
       END
```

## 総合問題5

**[演習5-1]** 5.1.1項で紹介したプログラムS1を、次のアドレスからロードした時のラベルX,Y,Z,Wのアドレスを16進数値で答えなさい。

(1) 0番地、(2) 1000番地、(3) 2000(16)番地

**[演習5-2]** レジスタGR1に1、GR2に2、GR3に3、GR4に4が格納されているとします。このとき、ラベルXが定義されている領域から5バイトの領域にある全ての語に100をセット(格納)する最も効率の良いと思われるプログラムを記述しなさい。ただし、レジスタGR0には、すでに100がセットされているとする。

**[演習5-3]** 次の記述に対して、正誤を判断し、誤りのあるものはその理由を述べよ。

(1) 汎用レジスタの値を32倍することは1命令ではできない。
(2) 汎用レジスタの値を60倍することは1命令ではできない。
(3) 「DC 1,2」も「DS 2」もともに2語記憶域を確保する。
(4) 指標レジスタとして使えるのは、GR0からGR7までの8個の汎用レジスタである。
(5) 「LD GR1,X,GR1」というように一つの命令の中で同一の汎用レジスタを2つ指定できない。
(6) 汎用レジスタの番号の少ない方がアクセス効率が良いので、良く使われる。
(7) リテラルを使うとアセンブラプログラムの記述量が少なくなり、読解性も向上するが、これは主記憶装置の節約にはならない。
(8) 「LAD GR1,100」によって生成される実効アドレスは100である。
(9) 「LD GR1,150」によって生成される実効アドレスは「150+GR0の値」である。
(10) 「LD GR2,100,GR3」によって生成される実効アドレスは、GR3が10ならば、110である。

# *Memorandum*

# 第6章 機械語命令の詳細

前章までは、わりと直感的に機械語命令を紹介してきました。本章では、すべての機械語命令を詳細に解説します。しかし、機械語の理解は、実際のCASL IIのプログラミングと密接に関係するので、最初に本章を読んだときに、本章のすべての内容を完全に理解する必要はありません。最初は本章をザーと流し読みして、第7章以降のプログラミングを中心とした章を読みながら、必要に応じて戻って読んで下さい。なお、本章ではマニュアル的な要素が強いので、演習問題はありません。

さて、機械語には、次のものがあります。

表 6-1 機械語命令の概要とその説明個所

| 種　類 | 説明個所 | 概　要 |
|---|---|---|
| ロード命令 | 6.1.1 | 記憶装置→レジスタ、レジスタ→レジスタのデータ転送(複写) |
| ストア命令 | 6.1.2 | レジスタ→記憶装置のデータ転送(複写) |
| ロードアドレス命令 | 6.1.3 | 実効アドレス→レジスタ |
| 算術加減算命令 | 6.2 | 符号付きデータの加減算 |
| 論理加減算命令 | 6.2 | 符号無しデータの加減算 |
| 論理演算命令 | 6.3 | ビット毎の論理演算 |
| 比較演算命令 | 6.4 | データの比較を行う命令(符号付き用・符号無し用) |
| シフト演算命令 | 6.5 | シフト処理のための演算(符号付き用・符号無し用) |
| 分岐命令 | 6.6 | プログラムの実行の流れを条件によって変更する命令 |
| スタック操作命令 | 6.7 | スタックと呼ばれる記憶装置の一部を扱う命令 |
| コール・リターン命令 | 6.8 | サブルーチンに関する命令 |
| その他の命令 | 6.9 | その他 |

なお、各命令の具体的な機械語と命令語の長さ(語長)は、情報処理技術試験案内の「仕様」には示されていませんが、「参考資料」の中に示されています。本書の記述、および、本書で提供するソフトウェアはすべて、その「参考資料」に完全に準拠しています。

# 6.1 ロード、ストア、ロードアドレス命令

主記憶装置と汎用レジスタの間のデータ転送、あるいは汎用レジスタ間のデータ転送、生成された実効アドレスの汎用レジスタへの転送などのために、LD(ロード)、ST(ストア)、LAD(ロードアドレス)命令があります。

**表6-2 ロード、ストア、ロードアドレス命令**

| 命　令 | 書き方 命令コード | 書き方 オペランド | 命令の説明 | FRの設定 |
|---|---|---|---|---|
| ロード LoaD | LD | r1,r2 | r1 ← (r2) | OFは常に0 |
| | | r,adr[,x] | r ← (実効アドレス) | SF, ZFは設定される |
| ストア STore | ST | r,adr[,x] | 実効アドレス ← (r) | 設定されない |
| ロードアドレス Load ADress | LAD | r,adr[,x] | r ← 実効アドレス | 設定されない |

表の見方：(6章のすべての記述に共通)

「書き方」の説明

r, r1, r2は汎用レジスタ(GR0～GR7)を意味します。xは指標レジスタ(GR1～GR7)を意味します。adrは、ラベル、10進定数、16進定数、あるいはリテラル(第5章参照)を意味します。なお、adrに指定できる数値は0～65535(=$FFFF_{(16)}$)です。[ ]内はその中が省略できることを意味します。

「命令の説明」の説明

$\alpha \leftarrow \beta$は、データ$\beta$を$\alpha$に転送(複写)することを意味します。$\beta$は数値か生成された実効アドレスです。

$\alpha \leftarrow (\beta)$は、$\beta$の内容(すなわち、値)を$\alpha$に転送(複写)することを意味します。

したがって、

「r1←(r2)」は、汎用レジスタr2の内容を汎用レジスタr1に転送(複写)することを意味します。

「r←(実効アドレス)」は、実効アドレスの内容を汎用レジスタrに転送(複写)することを意味します。

「実効アドレス←(r)」は、汎用レジスタrの内容を、主記憶装置の実効アドレス番地の語に転送(複写)することを意味します。

「r←実効アドレス」は、実効アドレスそのものを汎用レジスタrに転送(複写)することを意味します。

## 6.1.1 ロード命令

それでは、ロード命令について詳しく見てみましょう。**ロード命令**(LD)とは、データの転送を行うための機械語ですが、表6-2にあるように次の2つのパターンがあります。

```
LD      r1,r2
LD      r,adr[,x]
```

最初の書き方「LD r1,r2」は汎用レジスタ間でのデータの転送です。たとえば、次のように書きます。

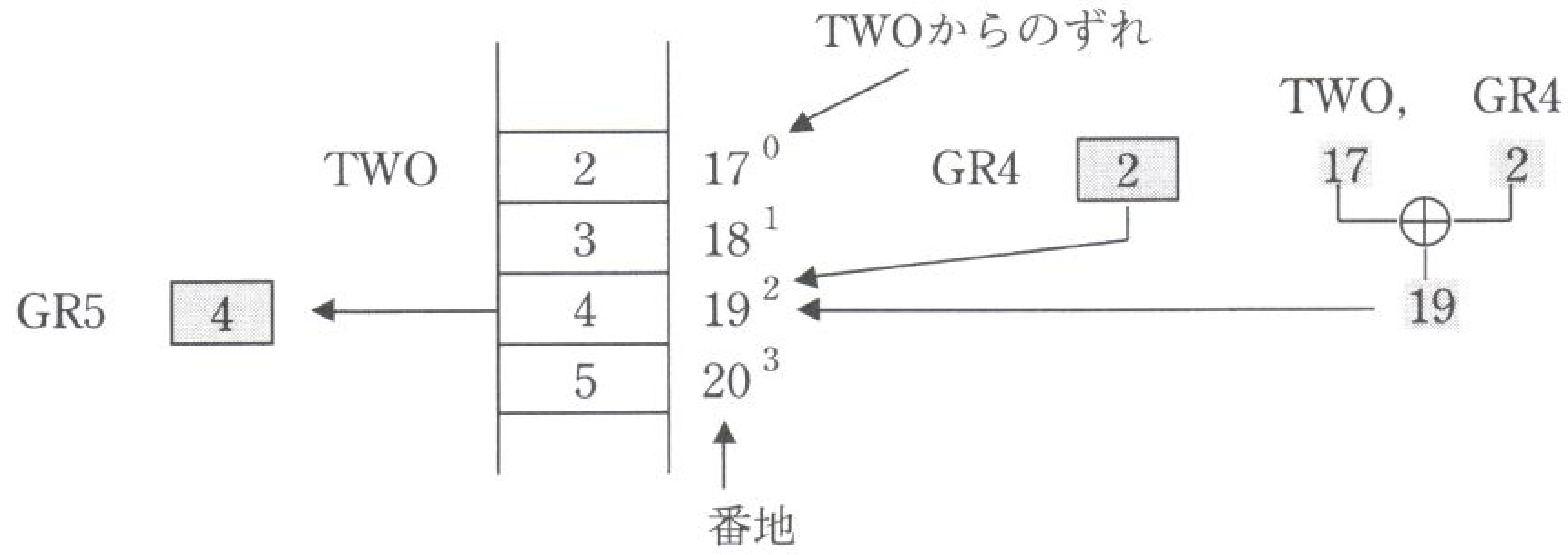
```
LD      GR0,GR1      GR1の内容をGR0にロードする
LD      GR7,GR0      GR0の内容をGR7にロードする
```

次の書き方「LD　r,adr[,x]」が、主記憶装置の内容を汎用レジスタに転送する場合の書き方です。xには、第5章で学習した指標レジスタを指定します。

これらの記述例を含んだプログラムLOADSAMPを以下に示します。

```
LOADSAMP START
         LD  GR0,ONE      ラベルONEの領域の値をGR0にロードする(LD r,adrの形式)
         LD  GR1,GR0      レジスタGR0の内容をGR1にロードする(LD r1,r2の形式)
         LD  GR2,TWO      ラベルTWOの領域の値をGR0にロードする(LD r,adrの形式)
         LD  GR3,=3       リテラル(3)の値をGR3へロードする(第5章参照)(LD r,adrの形式)
         LD  GR4,=2       リテラル(2)の値をGR4へロードする(第5章参照)(LD r,adrの形式)
         LD  GR5,TWO,GR4  「ラベルTWOの領域のアドレス+指標レジスタGR4の内容」番地の内容をGR5にロードする(LD r,adr,xの形式)
         LD  GR6,17       17番地(絶対番地)の内容をGR6にロードする(LD r,adrの形式)
         LD  GR7,17,GR4   「17番地(絶対番地)+GR4の内容」番地の内容をGR7にロードする(LD r,adr,xの形式)
         RET
ONE      DC  1
TWO      DC  2
         DC  3
         DC  4
         DC  5
         END
```

指標レジスタを使った命令「LD　GR5,TWO,GR4」は、TWOというラベルからちょうどGR4の値分だけ離れた番地の値をGR5に格納する命令であると理解することも出来ます。




したがって、このプログラムで「LD GR5,TWO,GR4」を実行すると、GR5の値は4になるはずです。生成される機械語とラベルのアドレスを確認するためにアセンブラリストを示します。

```
    Machine  Source List (LOADSAMP1.casl)
 Adr. Code Line          Program
     |    |  1| LOADSAMP START
    0|1000|  2|          LD   GR0,ONE       ラベルONEの領域の値をGR0にロードする(LD r,adrの形式)
    1|0010|   |
    2|1410|  3|          LD   GR1,GR0       レジスタGR0の内容をGR1にロードする(LD r1,r2の形式)
    3|1020|  4|          LD   GR2,TWO       ラベルTWOの領域の値をGR0にロードする(LD r,adrの形式)
    4|0011|   |
    5|1030|  5|          LD   GR3,=3        リテラル(3)の値をGR3へロードする(第5章参照)(LD r,adrの形式)
    6|0015|   |
    7|1040|  6|          LD   GR4,=2        リテラル(2)の値をGR4へロードする(第5章参照)(LD r,adrの形式)
    8|0016|   |
    9|1054|  7|          LD   GR5,TWO,GR4「ラベルTWOの領域のアドレス+指標レジスタGR4の内容」番地の
    A|0011|   |
     |    |  8|                              ;内容をGR5にロードする(LD r,adr,xの形式)
    B|1060|  9|          LD   GR6,17        17番地(絶対番地)の内容をGR6にロードする(LD r,adrの形式)
    C|0011|   |
    D|1074| 10|          LD   GR7,17,GR4   「17番地(絶対番地)+GR4の内容」番地の内容を
    E|0011|   |
     |    | 11|                              ;GR7にロードする(LD r,adr,xの形式)
    F|8100| 12|          RET
   10|0001| 13| ONE      DC   1
   11|0002| 14| TWO      DC   2
   12|0003| 15|          DC   3
   13|0004| 16|          DC   4
   14|0005| 17|          DC   5
   15|0003|   | (リテラルによって生成)
   16|0002|   | (リテラルによって生成)
     |    | 18|          END
```

リテラル「＝3」によって、15(16)番地に「DC 3」と書いたのと同じように値3が配置され、「LD GR3,＝3」が、「LD　GR3,0015」になっていることに注意してください(下のトレース参照)。この命令の実行によって、「15(16)番地の内容(3)」がGR3にセットされます。

また、9行目(LD　GR6,17)と10行目(LD　GR7,17,GR4)で記述している17番地というアドレスは、16進数では11(16)番地ですから、ちょうどTWOの領域のアドレスを指していることが分かります。この指定は、ラベルでなく直接数値で指定しているので、絶対番地になります。ラベルTWOの実行時のアドレスは、この機械語プログラムが主記憶装置にロードされるときの先頭アドレスによって変わります。TWOが17番地に配置されるためには、このプログラムは0番地にロードされなければなりません。それでは、0番地からロードしてこのプログラムを実行してみましょう。

```
 PR  Instruction       SP  OSZ GR0  GR1  GR2  GR3  GR4  GR5  GR6  GR7  Explanation
0000 LD   GR0,0010     7530---0001 C823 98BE E784 CAE1 BD6C ACD6 F2AE 0010:[h:0001, s:1, u:1]
0002 LD   GR1,GR0      7530---0001 0001 98BE E784 CAE1 BD6C ACD6 F2AE GR0:[s:1, u:1]
0003 LD   GR2,0011     7530---0001 0001 0002 E784 CAE1 BD6C ACD6 F2AE 0011:[h:0002, s:2, u:2]
0005 LD   GR3,0015     7530---0001 0001 0002 0003 CAE1 BD6C ACD6 F2AE 0015:[h:0003, s:3, u:3]
0007 LD   GR4,0016     7530---0001 0001 0002 0003 0002 BD6C ACD6 F2AE 0016:[h:0002, s:2, u:2]
0009 LD   GR5,0011,GR4 7530---0001 0001 0002 0003 0002 0004 ACD6 F2AE 0013:[h:0004, s:4, u:4]
000B LD   GR6,0011     7530---0001 0001 0002 0003 0002 0004 0002 F2AE 0011:[h:0002, s:2, u:2]
000D LD   GR7,0011,GR4 7530---0001 0001 0002 0003 0002 0004 0002 0004 0013:[h:0004, s:4, u:4]
000F RET               7530---0001 0001 0002 0003 0002 0004 0002 0004 Return to OS
```

「LD　GR7, 17, GR4」(上記のトレースでは「LD　GR7,0011,GR4」)は、17番地(11(16)番地)＋GR4の値、すなわち、19(＝13(16))番地の内容(4)をレジスタGR7にロードします。

また、プログラムLOADSAMPのアセンブラリストに示された機械語のアドレスからも分かるように、レジスタ間のロード命令「LD r1,r2」はその語長が1語ですが、主記憶装置と汎用レジスタとの間のロード命令「LD r,adr[,x]」はその語長が2語になっています。前述のように、これらの情報は、本書の最後に、付録として示されています。

### 6.1.2 ストア命令

**ストア命令**(ST)は、汎用レジスタの値を主記憶装置に転送する「ST　r,adr[,x]」の形式のみで汎用レジスタ間に対するものはありません。これは、adr[,x]で生成される実効アドレスの語に、汎用レジスタrの内容を転送(複写)する命令です。

いくつか例を示します。

```
    ST  GR0,100        GR0の値を100番地(絶対番地)に転送(複写)する。
    ST  GR1,BAR        GR1の値をラベルBARの領域に転送(複写)する。
    ST  GR2,100,GR4    GR2の値を「100＋GR4の内容」番地に転送(複写)する。
    ST  GR3,BAR,GR5    GR3の値を「ラベルBARの領域のアドレス＋GR5の内容」番地に転送(複写)する。
     :
BAR DS  5
```

なお、上記のプログラムで「BAR　DS　5」としていますが、これは単に連続した5語の領域を確保することが目的なだけで、プログラムの動作の観点からは、たとえば、次のように記述しても全く問題はありません。

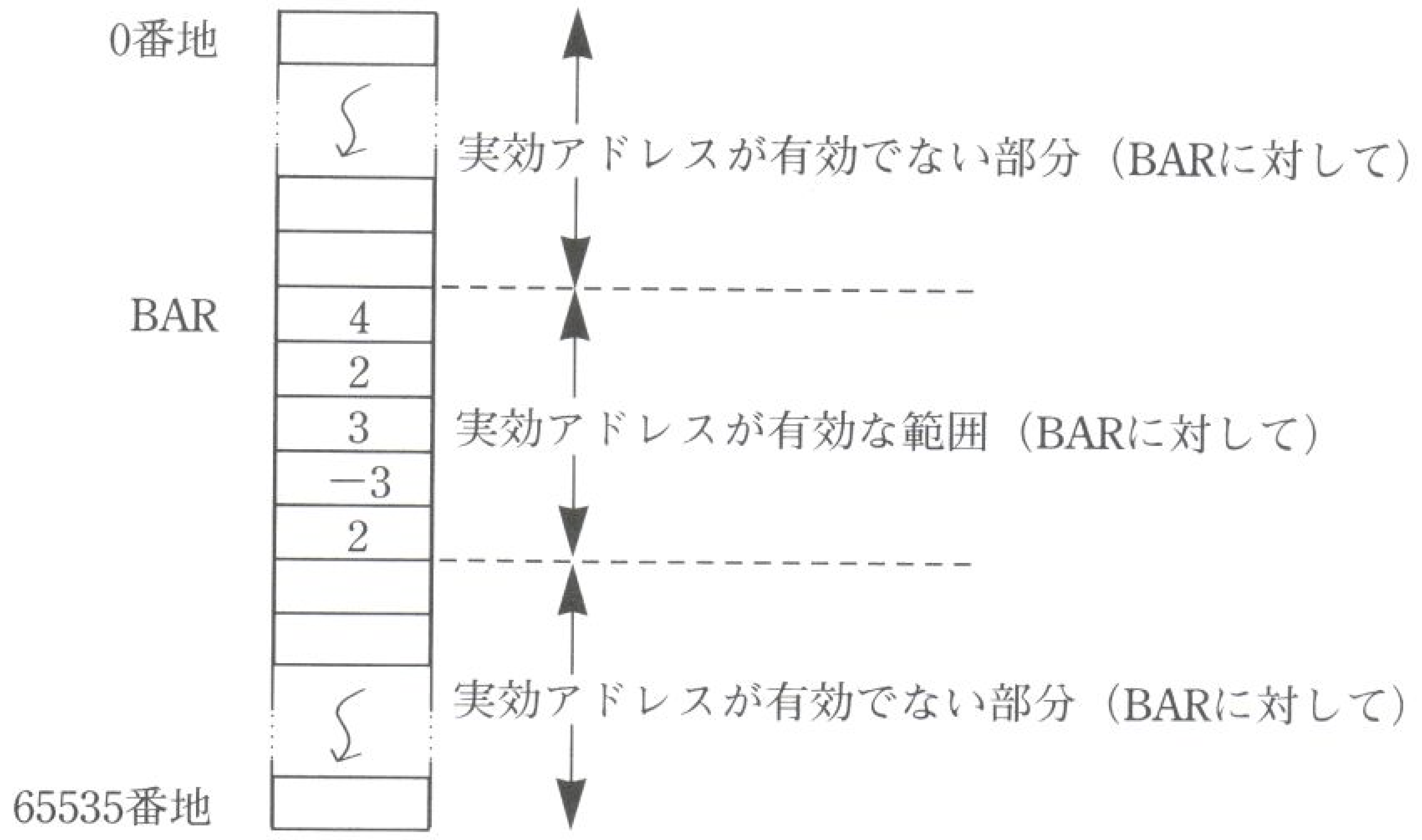
```
BAR     DC      4
        DC      2
        DC      3
        DC     −3
        DC      2
```

あるいは

```
BAR     DC        4,2,3,−3,2
```

この記述によって、BAR以降に連続した5語が確保されます。もちろん、この例では、上記の値がその領域に設定されますが、これらの値をプログラムで利用しなくても問題ありません。DS命令の場合には、領域が確保されるだけで、値は設定されません。

また、指標レジスタを使う場合、生成された実効アドレスが「有効なアドレス」でなくてはなりません。有効なアドレスとは、その目的のために用意された領域内のアドレスのことです。この例では、BARの領域は5語分になっています。




したがって、「ST GR3,BAR,GR5」としたとき、指標レジスタGR5の値が0〜4の場合であれば、生成される実効アドレスは、その範囲の中で有効なアドレスとなりますが、たとえば、GR5の値が100の場合、「ST　GR3,BAR,GR5」を実行すると領域BARの範囲外の領域に値をストアしてしまいます。もし、これが機械語命令の部分であれば、プログラムの実行が正しく行われなくなり変なことが起こることがあります。

この「変なことが起こること」について詳しく説明しましょう。以下のプログラムは一見何の変わりもない単純なプログラムです。

```
BUG1    START
        LD      GR0,=#8100
        LD      GR1,=−3
        ST      GR0,TABLE,GR1    ←「悪さ」をする命令
        LD      GR2,=#CAFE
        RET
TABLE   DC      1,2,3,4,5
        END
```

アセンブラリストから分かるように、TABLEのアドレスは9番地です。ここで、「ST　GR0, TABLE, GR1」で何が起こるか考えてください。GR1にはその直前で−3が入っています。したがって、実効アドレスは、9＋(−3)＝6番地になります。つまり、6番地に8100(16)が格納されます。ところが、6番地には何が入っているかというと、「LD GR2,=#CAFE」という命令の第1語目が入っています。

何と、4行目のST命令がこのLD命令の第1語目の値(1020(16))を8100(16)に変えてしまうのです。8100(16)という数値は、「RET」命令の機械語に対応しています。したがって、「ST GR0, TABLE, GR1」を実行後、6番地を実行しようとすると、そこには書き直された値8100(16)、すなわちRET命令の機械語が入っていて、それが実行されてしまいます。

COMET II(に限らず一般的にコンピュータの制御装置)は、6番地の8100(16)が誤ってデータとして書き直されてしまったことを知る由もありません。COMET IIは、主記憶装置上にある数値を機械語だと思って淡々と実行する単純な装置(マシン)に過ぎないのです(＊1)。

(＊1) ただし、本書に付属のCASLDVは、初心者教育用なので、利用者のこのようなミスを発見して、エラーとして報告してくれます。ここに示されているトレースの結果(実行結果)は、そのチェックをはずしているので、そのまま、実行されています。このチェックを外すには、CASLDVの場合には、メニューから「オプション」を選び、さらに「実行時のオプションの指定」を選び、「さまざまな実行時オプション」グループの中から「記憶領域の境界のチェックを行う」チェックマークを外してください。コマンドライン版をお使いの方は、comet_IIコマンドに-vオプションを付けてください。

```
     Machine        Source List (bug1.casl)
  Adr.  Code    Line    Program
                  1  BUG1    START
     0  1000      2          LD   GR0,=#8100
     1  000E
     2  1010      3          LD   GR1,=－3
     3  000F
     4  1101      4          ST   GR0,TABLE,GR1  ←この命令で6番地の内容(もともとは1020(16))を8100(16)にしてしまう
     5  0009
     6  1020      5          LD   GR2,=#CAFE
     7  0010
     8  8100      6          RET
     9  0001      7  TABLE   DC   1,2,3,4,5
     A  0002
     B  0003
     C  0004
     D  0005
     E  8100         (リテラルによって生成)
     F  FFFD         (リテラルによって生成)
    10  CAFE         (リテラルによって生成)
                  8          END
```

それでは、実際に実行してみましょう。

```
  PR Instruction       SP  OSZ  GR0  GR1  GR2  GR3  GR4  GR5  GR6  GR7 Explanation
0000 LD  GR0,000E     7530 -S-  8100 C823 98BE E784 CAE1 BD6C ACD6 F2AE 000E:[h:8100, s:-32512, u:33024]
0002 LD  GR1,000F     7530 -S-  8100 FFFD 98BE E784 CAE1 BD6C ACD6 F2AE 000F:[h:FFFD, s:-3, u:65533]
0004 ST  GR0,0009,GR1 7530 -S-  8100 FFFD 98BE E784 CAE1 BD6C ACD6 F2AE GR0:[h:8100, s:-32512, u:33024]
0006 RET              7530 -S-  8100 FFFD 98BE E784 CAE1 BD6C ACD6 F2AE Return to OS
```

「ST」命令実行後、6番地でRETが実行されていることが分かります。ST命令は、主記憶装置の内容を書き直すことができるので、指標レジスタの範囲を間違えるとこのようなことが起きてしまいます。

ただし、上級プログラマは、逆にこの機能を使って、自分で自分のプログラムを変更するという高度なプログラムを書くこともあります。

### 6.1.3 ロードアドレス命令

本項では、**ロードアドレス命令**(LAD)という一風変わった、しかし、非常に利用価値の高い命令を紹介します。これは、次の処理を行うために使います。

(1) 汎用レジスタに定数を設定する
(2) 汎用レジスタにある一定の値を加えたり、汎用レジスタからある一定の値を減じる
(3) 汎用レジスタ間で簡単な加減算を行う
(4) ラベルのアドレスを求める

以上のようなさまざまな処理が出来ます。ただし、LAD命令では、フラグレジスタの値は変化しま

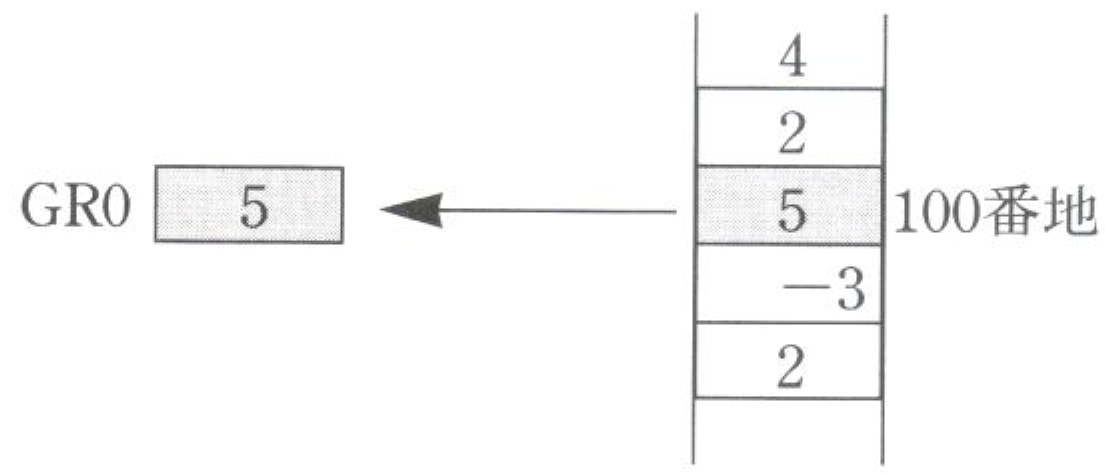
せん(*1)。フラグレジスタの値を変更させたい場合には、LAD命令ではなく、LD, ADDA, SUBA命令などを使います。これについては、6.2節で詳しく説明します。

LAD命令は、次の形式をしています。

```
LAD      r,adr[,x]
```

これは、adr[,x]の部分で生成される実効アドレスそのものを汎用レジスタrに格納するという命令です。しかし、この説明では、何を言っているか分からないでしょう。そこで例を用いて説明します。まず、LD命令をちょっと復習しましょう。

LD　　GR0, 100




「LD　GR0,100」は、100番地(絶対アドレス)の値をレジスタGR0にロードする命令でした。ここで生成された実効アドレスは、100(番地)ですね。

#### (1) 汎用レジスタに値を設定する

それでは、次のLAD命令を見てください。

LAD　　GR0, 100




この命令で生成される実効アドレスも100ですね。LD命令とは異なり、LAD命令は、生成されたこの実効アドレスそのものを汎用レジスタに格納するのです。したがって、GR0には100が格納されます。これは、とても便利です。なぜなら、実効アドレスをロードするという本来の機能を利用して、定数をそのまま、汎用レジスタに格納することができるのです。

たとえば、次の命令を見てください。

```
LAD      GR1,-1
LAD      GR2,#CAFE
```

(*1) COMET IIの以前のバージョン(COMET)では、LAD命令に相当する命令としてLEA命令がありました。これは、フラグレジスタの値を変化させましたが、COMET IIのLAD命令では、フラグレジスタの値は変化しません。この理由は、おそらく、ADDAやSUBAなどの命令で汎用レジスタ間の演算がCOMET IIから可能になり、LAD命令でもフラグレジスタが変化してしまうと、情報処理技術者試験の問題の解答欄にLAD命令とADDA命令の2種類の選択肢が出来てしまうからであると類推できます。

上記の命令を実行すると、GR1には－1が、GR2にはCAFE(16)が格納されます。

ところで、定数を格納するには、リテラルを使う方法もありました。たとえば、同じ値をリテラルでセットするには、次のようにします。

```
LD      GR1, =－1
LD      GR2, =#CAFE
```

この方法と、LADによる方法の違いについては、7.1.1項で説明します。

### (2) 汎用レジスタにある一定の値を加えたり、汎用レジスタからある一定の値を減じる

LAD命令を使えば、汎用レジスタ間の簡単な計算もできます。これもまた、実効のアドレスの生成の仕組みを転用しています。

GR1に100が入っているとします。その状態で次の命令を実行すると、GR1の値は101になります。つまり、GR1の値を1つカウントアップすることができます。

```
LAD     GR1,1,GR1
```

これは、次の仕組みで実現されています。この命令の実効アドレスは、「1＋GR1の内容」でした。GR1の値は100ですから、「1＋100」＝101が実効アドレスとなります。したがって、この値が、GR1にセットされます。

```
LAD     GR1,1,GR1
        ↑   ↓  ↓
        |   1+100 = 101
        └──────────┘
```

この機能によって、たとえば、次の演算が可能になります。

```
LAD     GR2,－1,GR2        GR2を1つカウントダウン
LAD     GR3,15,GR3         GR3に15を加える
LAD     GR4,#2BCD,GR4      GR4に2BCD(16)を加える
LAD     GR5,#FFFF,GR5      GR5を1つカウントダウン
```

最後の例では、1つカウントダウンとなっています。これは、FFFF(16)が－1を意味するからです。6.2.3項で説明しますが、実効アドレスを生成するときの演算は、論理加算と呼ばれる(符号無しの)演算が行われます。しかし、論理加算であっても、実際には、汎用レジスタに、FFFF(16)を加えたり、(FFFF(16)の符号付き表現である)－1を加えても、符号付きの計算と同じ結果となります。この辺の仕組みついては、2.6.3項で詳しく解説しました。

なお、ここでちょっと注意が必要です。指標レジスタとして使えるのは、汎用レジスタのGR1～GR7であって、GR0は使えません。つまり、次の計算はできませんので、注意が必要です。

```
LAD     GR0,1,GR0  間違い！(GR0は指標レジスタとしては使えない)
                ↑
        指標レジスタにGR0は使えない
```

したがって、汎用レジスタに対して定数値を加減算できるといっても、GR0は使用できないので注意が必要です。ただし、LAD命令で定数を代入する場合には、指標レジスタではないので使用できます。

```
LAD      GR0,100      正しい使い方
          ↑
          これは指標レジスタではないので記述OK!
```

**ポイント**

GR0は、指標レジスタとしては使用できない。

### (3) 汎用レジスタ間で簡単な加減算を行う

(2)と同じ仕組みで、汎用レジスタ間の簡単な計算を行うことができます。

```
LAD      GR0,－2,GR1        GR0 ← GR1 － 2
LAD      GR1,1,GR2          GR1 ← GR2 ＋ 1
LAD      GR2,#CAFE,GR5      GR2 ← GR5 ＋ CAFE(16)
```

一番、最初の例を図示すると、次のようになります。GR1には100が格納されているとします。

```
LAD      GR0,－2,GR1
          ↑    ↓   ↓
          |    －2+100 = 98
          └──────────────┘
```

上記のように、GR0には、GR1－2の値である98が格納されます。

### (4) ラベルのアドレスを汎用レジスタに設定する

LAD命令の本来の目的は、実効アドレスを汎用レジスタに設定することです。つまり、「ラベルのアドレスを汎用レジスタに設定する」ことが本来のLAD命令の機能です。

次の例を見てください。

```
          LAD      GR0,TABLE
           :
TABLE     DC       2,3,4,5,6,7,8,9
```

このLAD命令の実行によって、レジスタGR0には、ラベルTABLEのアドレスが格納されます。

指標アドレスと組み合わせて使うと簡単に各データのアドレスを知ることが出来ます。説明のために、紙面は必要になりますが、DC命令を8行に分けて示すことにします。

```
         LAD      GR0,TABLE,GR7
         :
TABLE    DC       2  ← GR7=0のとき、GR0に設定されるアドレス
         DC       3  ← GR7=1のとき、GR0に設定されるアドレス
         DC       4  ← GR7=2のとき、GR0に設定されるアドレス
         DC       5  ← GR7=3のとき、GR0に設定されるアドレス
         DC       6  ← GR7=4のとき、GR0に設定されるアドレス
         DC       7  ← GR7=5のとき、GR0に設定されるアドレス
         DC       8  ← GR7=6のとき、GR0に設定されるアドレス
         DC       9  ← GR7=7のとき、GR0に設定されるアドレス
```

イメージを具体的にするために、実際にプログラムを動かしてみましょう。次のプログラムは、レジスタGR7に4を格納し、「6」が格納された領域のアドレスをGR1に求め、さらにLD命令で、その番地の内容をGR2にロードしています。「LD GR2,0,GR1」では、生成される実効アドレスは「0＋GR1の値」ですから、「GR1の値」番地が実効アドレスになります。そのアドレスの内容(値)がGR2にロードされます。

```
LADTBL   START
         LAD      GR7,4                GR7 ← 4
         LAD      GR1,TABLE,GR7        GR1 ← 「TABLE番地＋GR7番地」
         LD       GR2,0,GR1            GR2 ← 「GR1のアドレスの内容」
         RET
TABLE    DC       2,3,4,5,6,7,8,9
         END
```

アドレスを知るために、アセンブラリストを示します。

```
  Machine            Source List (LADTBL.casl)
Adr.    Code Line              Program
   |      |  1 | LADTBL   START
  0| 1270 |  2 |          LAD  GR7,4           GR7 ← 4
  1| 0004 |    |
  2| 1217 |  3 |          LAD  GR1,TABLE,GR7   GR1 ← 「TABLE番地＋GR7番地」
  3| 0007 |    |
  4| 1021 |  4 |          LD   GR2,0,GR1       GR2 ← 「GR1のアドレスの内容」
  5| 0000 |    |
  6| 8100 |  5 |          RET
  7| 0002 |  6 | TABLE    DC   2,3,4,5,6,7,8,9
  8| 0003 |    |
  9| 0004 |    |
  A| 0005 |    |
  B| 0006 |    |
  C| 0007 |    |
  D| 0008 |    |
  E| 0009 |    |
   |      |  7 |          END
```

この例では、GR7に4を入れていますので、3行目のLAD命令では、値6が入っている領域のアドレス($B_{(16)}$)がGR1に格納されるはずです。そして、GR2には、その値6が格納されるはずです。

アセンブラリストとアドレスを一致させるため、0番地からロードして実行してみましょう。

```
 PR Instruction        SP  OSZ GR0  GR1  GR2  GR7  Explanation
0000 LAD  GR7,0004     7530 --- 8029 C823 98BE 0004 [h:0004, s:4, u:4]
0002 LAD  GR1,0007,GR7 7530 --- 8029 000B 98BE 0004 [h:000B, s:11, u:11]
0004 LD   GR2,0000,GR1 7530 --- 8029 000B 0006 0004 000B:[h:0006, s:6, u:6]
0006 RET               7530 --- 8029 000B 0006 0004 Return to OS
```

ちゃんと、レジスタGR1には$B_{(16)}$が、レジスタGR2には6が格納されています。

## 6.2 加算・減算演算命令

主記憶装置と汎用レジスタの間の演算(加算・減算)、あるいは汎用レジスタ間の演算(加算・減算)、を行うために、ADDA, SUBA, ADDL, SUBL命令があります。

**表6-3 加算・減算演算命令**

| 命 令 | 書き方 | | 命令の説明 | FRの設定 |
|---|---|---|---|---|
| | 命令コード | オペランド | | |
| 算術加算<br>ADD Arithmetic | ADDA | r1,r2 | r1 ← (r1)＋(r2) | OF, SF, ZFが設定される |
| | | r,adr[,x] | r ← (r)＋(実効アドレス) | |
| 算術減算<br>SUBtract Arithmetic | SUBA | r1,r2 | r1 ← (r1)－(r2) | OF, SF, ZFが設定される |
| | | r,adr[,x] | r ← (r)－(実効アドレス) | |
| 論理加算<br>ADD Logical | ADDL | r1,r2 | r1 ← (r1)＋(r2) | OF, SF, ZFが設定される |
| | | r,adr[,x] | r ← (r)＋(実効アドレス) | |
| 論理減算<br>SUBtract Logical | SUBL | r1,r2 | r1 ← (r1)－(r2) | OF, SF, ZFが設定される |
| | | r,adr[,x] | r ← (r)－(実効アドレス) | |

上記の表にあるように、演算には、算術演算と論理演算の2種類があります。簡単に言ってしまえば、**算術演算**は、対象データを符号付き(－32768～32767)とみなして計算する演算です。一方、**論理演算**は、対象データを符号無し(0～65535)とみなして計算する演算です。前節までの、加算・減算命令は、すべて算術演算を使ってきました。一方、アドレスの計算(指標レジスタを用いた実効アドレスの生成時の計算)はすべて論理演算です。この2つの違いは、何なのでしょう。この違いを調べるために、さまざまなデータの算術演算、論理演算を行ったときの表を以下に示します。

**表6-4 さまざまなデータの算術演算、論理演算を行ったときの表**(＊1)

| 16進数 | | 符号付き | | | | | | 符号無し | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 10進数 | | 算術演算の結果 | | | | 10進数 | | 論理演算の結果 | | | |
| X | Y | X | Y | 加算(X+Y) | OSZ | 減算(X−Y) | OSZ | X | Y | 加算(X+Y) | OSZ | 減算(X−Y) | OSZ |
| 0001 | 0001 | 1 | 1 | 2(0002) | --- | 0(0000) | --Z | 1 | 1 | 2(0002) | --- | 0(0000) | --Z |
| 0001 | FFFF | 1 | -1 | 0(0000) | --Z | 2(0002) | --- | 1 | 65535 | 0(0000) | O-Z | 2(0002) | O-- |
| CCCC | DDDD | -13108 | -8739 | -21847(AAA9) | -S- | -4369(EEEF) | -S- | 52428 | 56797 | 43689(AAA9) | OS- | 61167(EEEF) | OS- |
| 7FFF | 00FF | 32767 | 255 | -32514(80FE) | OS- | 32512(7F00) | --- | 32767 | 255 | 33022(80FE) | -S- | 32512(7F00) | --- |
| 8100 | 0FFF | -32512 | 4095 | -28417(90FF) | -S- | 28929(7101) | O-- | 33024 | 4095 | 37119(90FF) | -S- | 28929(7101) | --- |
| 0ABC | 0DEF | 2748 | 3567 | 6315(18AB) | --- | -819(FCCD) | -S- | 2748 | 3567 | 6315(18AB) | --- | 64717(FCCD) | OS- |

SZは、フラグレジスタの値を示します。また、－21847（AAA9）という表現は10進数では－21847、16進数ではAAA9ということを意味します。

この結果を見てみると、演算結果に示されている16進数の値(かっこの中)は、符号付きであっても符号無しであっても、まったく同じになっていることがわかります。しかし、符号の有無の違いが分かる10進数でみると、その値の解釈が異なり、それに伴って、フラグレジスタにセットされる内容が異なっていることに注意してください。

加算・減算命令はすべて、汎用レジスタ間の演算と、汎用レジスタと主記憶装置(実効アドレスで指定)間の演算に分かれます。

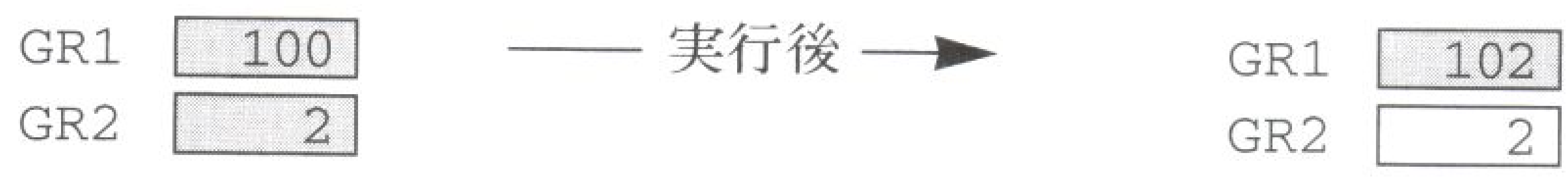
(＊1) この表6-4を確認するプログラムがCD-ROMのSourceフォルダの中のchap6フォルダのFRCHK.caslにありますので参考にして下さい。

## 6.2.1 算術加算

それでは、**算術加算**(ADDA)命令から例示しましょう。

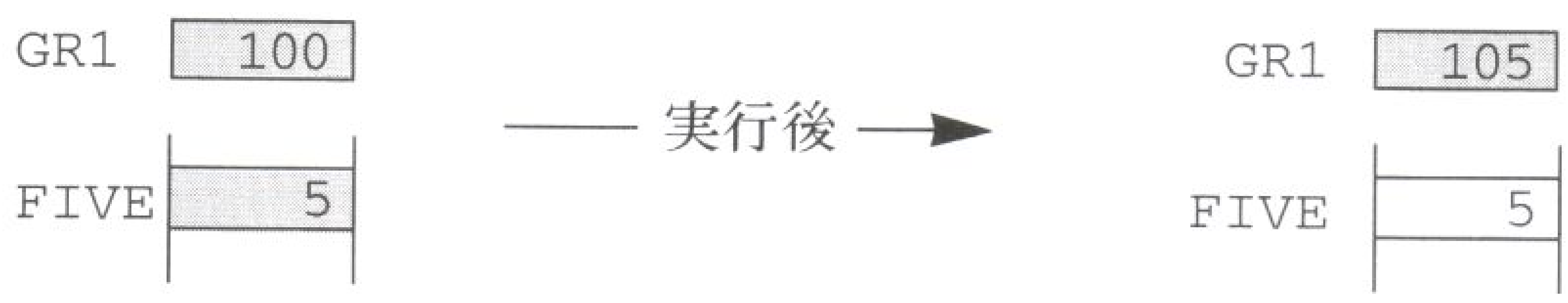
```
ADDA    GR1,GR2
```

GR1←GR1＋GR2




```
ADDA    GR1,FIVE
```

GR1←GR1＋FIVEの領域の値



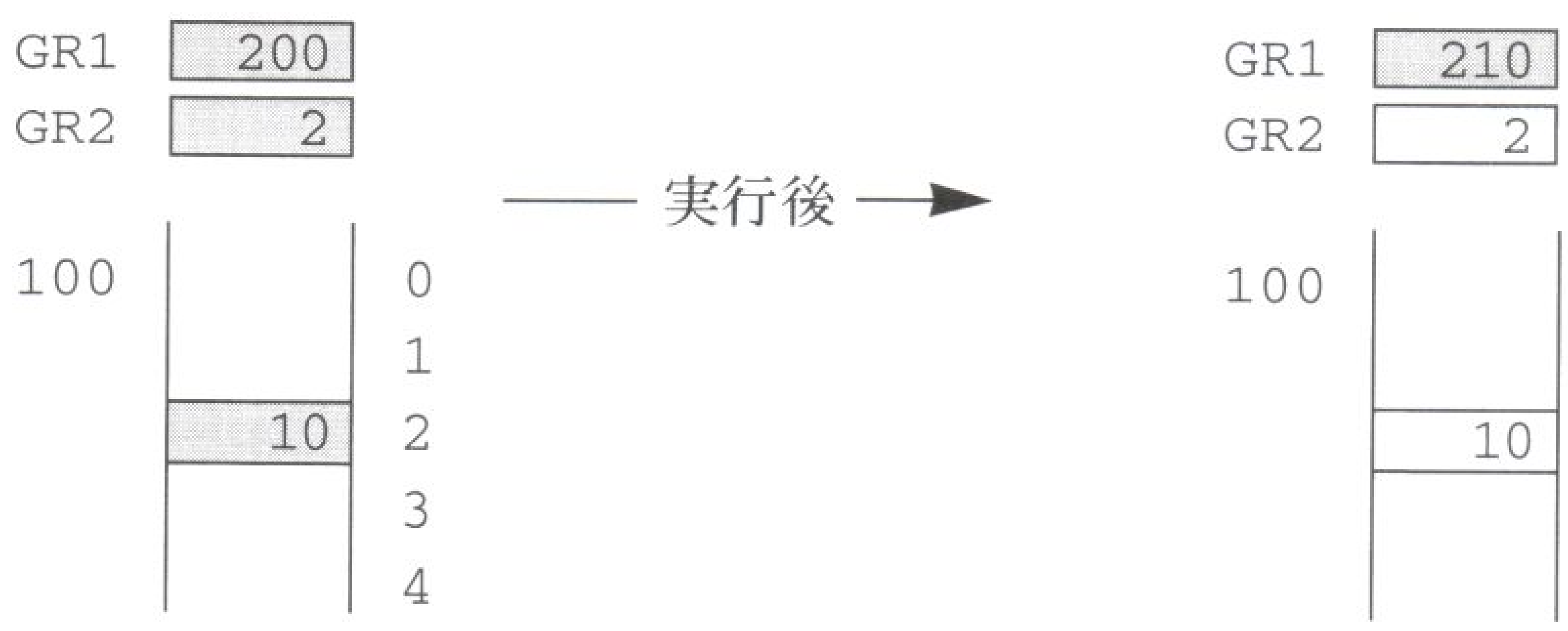

```
ADDA    GR1,TABLE,GR2
```

GR1←GR1＋「TABLE＋GR2」番地の領域の値



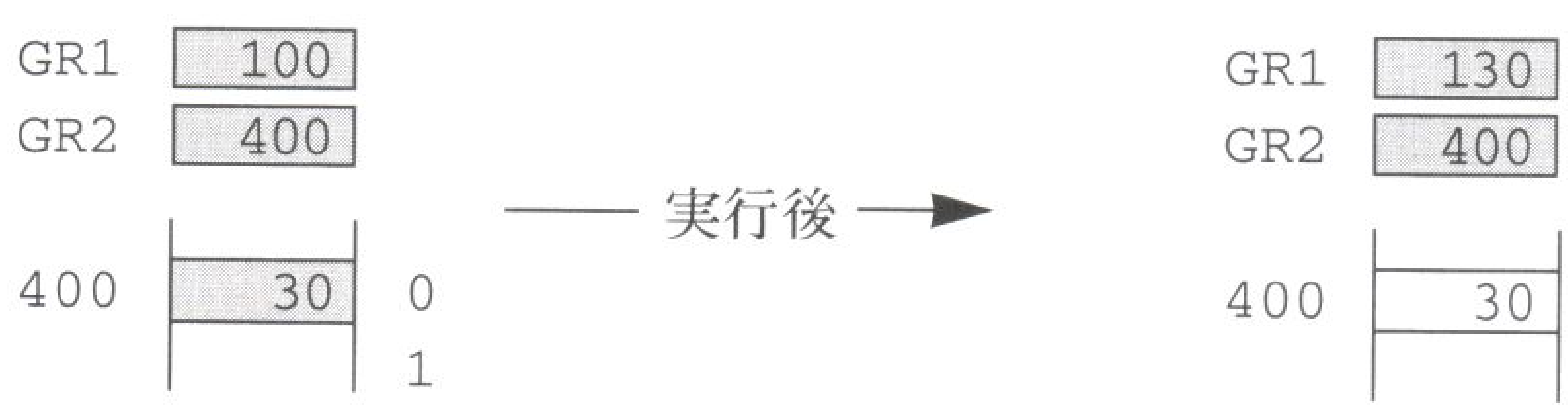

```
ADDA    GR1,100,GR2
```

GR1←GR1＋「100番地＋GR2」番地の領域の値(100は絶対番地)




```
ADDA    GR1,0,GR2
```

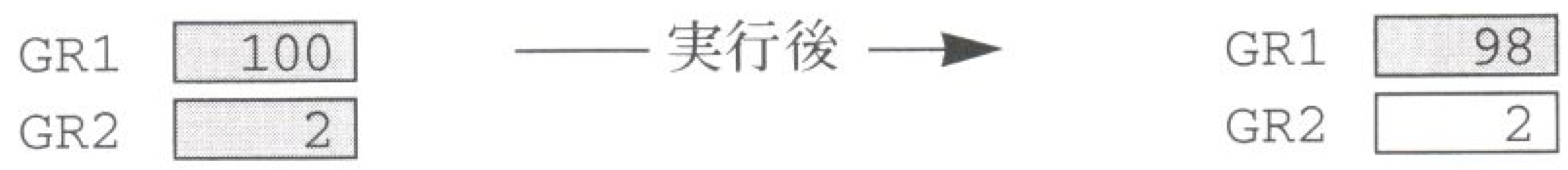
GR1←GR1＋「GR2番地の領域の値」



### 6.2.2 算術減算

次にSUBA命令を例示します。符号付きデータの減算になる点を除けばADDAと同じです。

```
SUBA    GR1,GR2
```
GR1←GR1－GR2



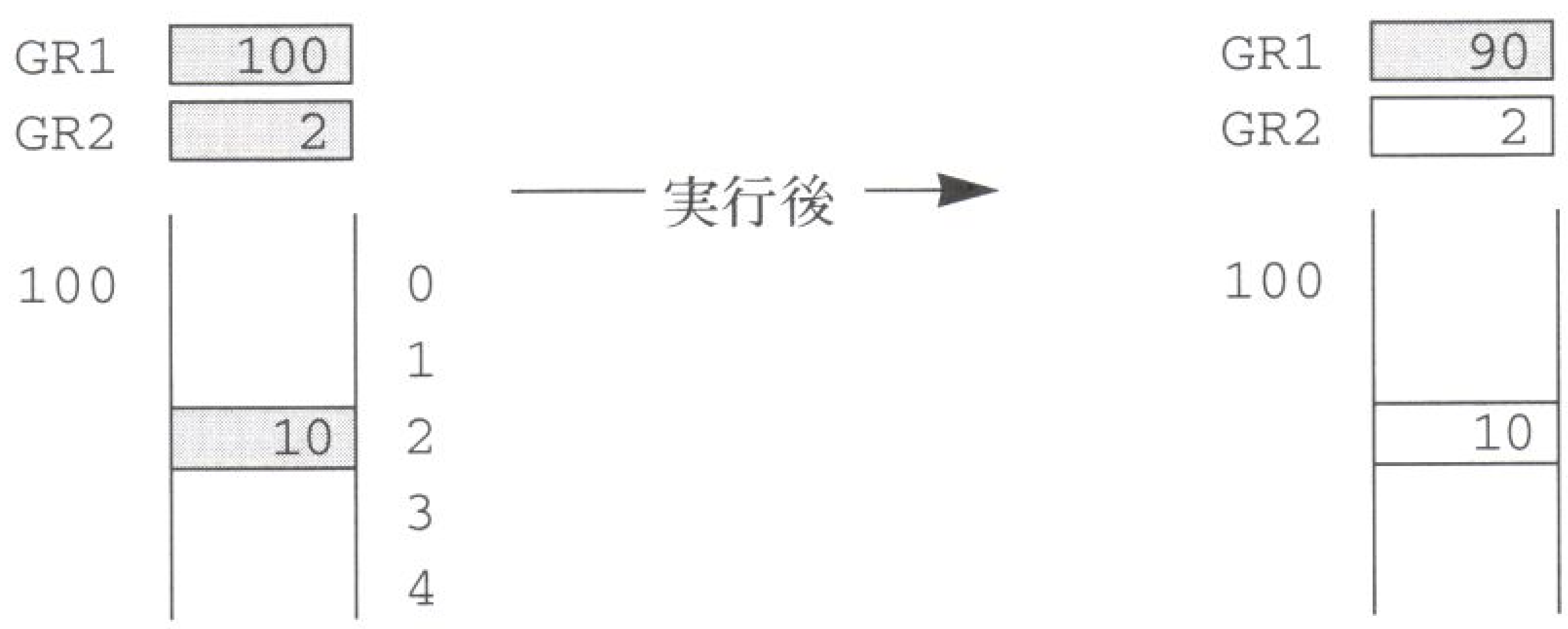

```
SUBA    GR1,FIVE
```
GR1←GR1－FIVEの領域の値



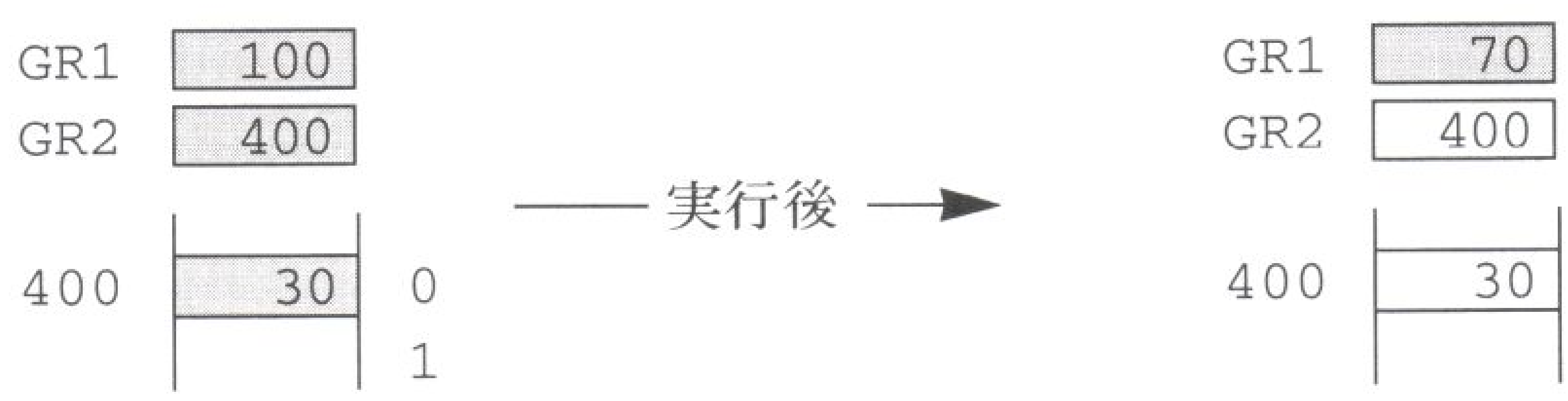

```
SUBA    GR1,TABLE,GR2
```
GR1←GR1－「TABLE＋GR2」番地の領域の値




```
SUBA    GR1,100,GR2
```
GR1←GR1－「100番地＋GR2」番地の領域の値（100は絶対番地）




```
SUBA    GR1,0,GR2
```
GR1←GR1－「GR2番地の領域の値」

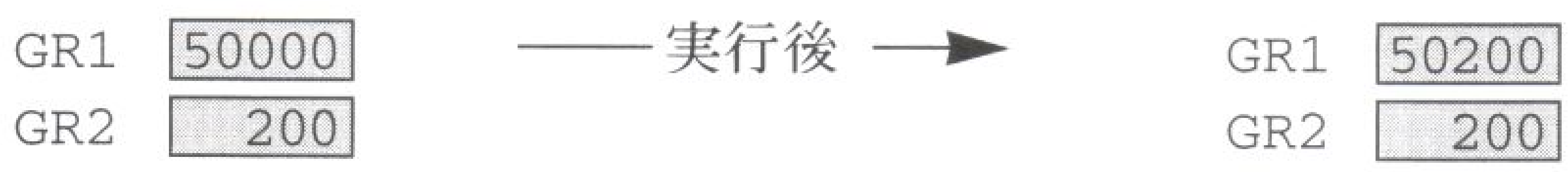


### 6.2.3 論理加算

それでは、今度は、論理加算・論理減算について、見てみましょう。基本的な考え方は、ADDA, SUBAと同じです。符号無しデータの加算になる点を除けばADDAと同じです。

**論理加算**(ADDL)命令を例示します。

`ADDL    GR1,GR2`　　GR1←GR1＋GR2




`ADDL    GR1,FIVE`　　GR1←GR1＋FIVEの領域の値



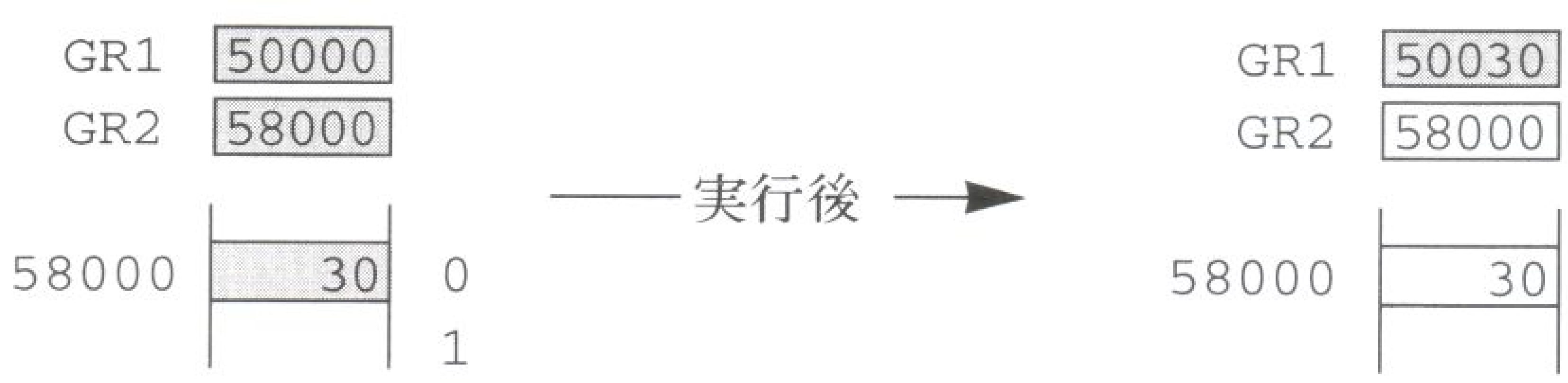

`ADDL    GR1,TABLE,GR2`　　GR1←GR1＋「TABLE＋GR2」番地の領域の値




`ADDL    GR1,100,GR2`　　GR1←GR1＋「100番地＋GR2」番地の領域の値 (100は絶対番地)




`ADDL    GR1,0,GR2`　　GR1←GR1＋「GR2番地の領域の値」

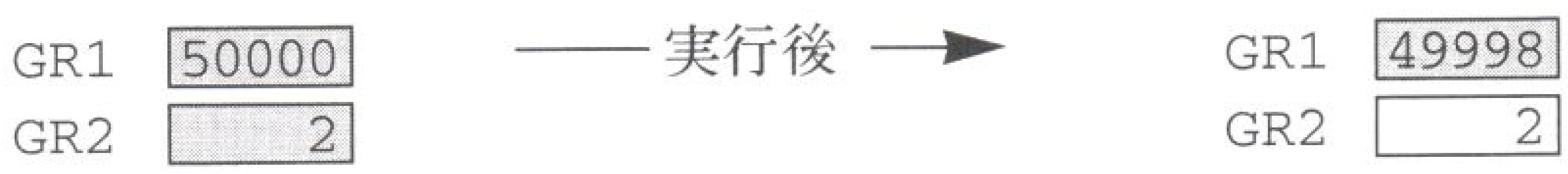


### 6.2.4 論理減算

次に**論理減算**(SUBL)命令を例示します。符号無しデータの減算になる点を除けばSUBAと同じです。

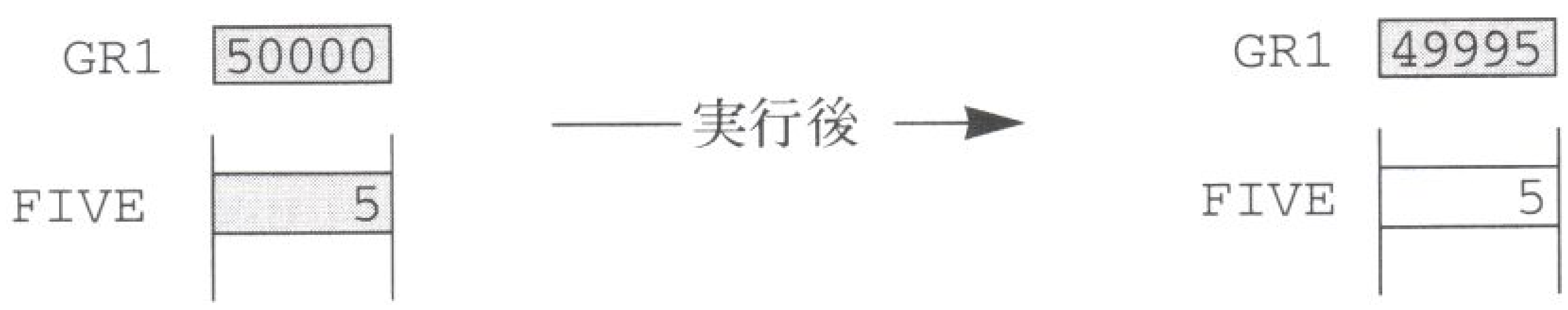
```
SUBL     GR1,GR2
```

GR1←GR1－GR2




```
SUBL     GR1,FIVE
```

GR1←GR1－FIVEの領域の値




```
SUBL     GR1,TABLE,GR2
```

GR1←GR1－「TABLE＋GR2」番地の領域の値




```
SUBL     GR1,100,GR2
```

GR1←GR1－「100番地＋GR2」番地の領域の値(100は絶対番地)




```
SUBL     GR1,0,GR2
```

GR1←GR1－「GR2番地の領域の値」



## 6.3 論理演算命令

**論理演算**には、**論理積**(AND)・**論理和**(OR)・**排他的論理和**(exclusive OR)の3つがあります。これらの命令は、主記憶装置と汎用レジスタの間の論理演算、あるいは汎用レジスタ間の論理演算を行います。実際の機械語命令には、AND命令(論理積), OR命令(論理和), XOR命令(排他的論理和)があります。

論理演算では、たとえば、次の処理を行うことができます。

(1) 特定のビットを抽出する(AND命令)。
(2) 特定のビットをセットする(OR命令)。
(3) 偶数か奇数かを判定する(AND命令)。
(4) 符号を判定する(AND命令)。
(5) 数値をゼロクリアする(XOR命令)。
(6) ビットを反転する(XOR命令)

**表6-5 論理演算命令**

| 命 令 | 書き方 | | 命令の説明 | FRの設定 |
|---|---|---|---|---|
| | 命令コード | オペランド | | |
| 論理積 AND | AND | r1,r2 | r1 ← (r1)AND(r2) | OFは常に0<br>SF, ZFは設定される |
| | | r,adr[,x] | r ← (r)AND(実効アドレス) | |
| 論理和 OR | OR | r1,r2 | r1 ← (r1)OR(r2) | OFは常に0<br>SF, ZFは設定される |
| | | r,adr[,x] | r ← (r)OR(実効アドレス) | |
| 排他的論理和 eXclusive OR | XOR | r1,r2 | r1 ← (r1)XOR(r2) | OFは常に0<br>SF, ZFは設定される |
| | | r,adr[,x] | r ← (r)XOR(実効アドレス) | |

論理演算とは、各データのビットに対する演算です。6-6表に1ビットの場合の各論理演算を示します。

**表6-6 1ビットごとの論理演算**

| AND | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| OR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| XOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

表6-6に示すように、0 AND 0 → 0、1 AND 0 → 0、0 AND 1 → 0、1 AND 1 → 1、となります。つまり、論理積は、どちらのビットも1のときだけ1となります。

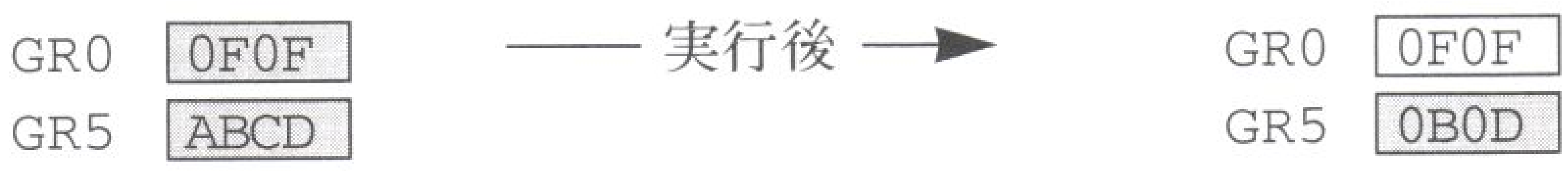
これに対して、論理和は、0 OR 0 → 0、1 OR 0 → 1、0 OR 1 → 1、1 OR 1 → 1、となります。つまり、論理和は、どちらかのビットが1であれば1となります。

排他的論理和は、0 XOR 0 → 0、1 XOR 0 → 1、0 XOR 1 → 1、1 XOR 1 → 0、となります。つまり、排他的論理和は、ビットのON/OFFが異なれば1となり、同じ場合には0となります。

### 6.3.1 論理積

AND命令は論理積を求めます。

```
AND      GR5,GR0
```




この演算では、ABCD　AND　0F0F の演算を行い、その結果を再びGR5にセットしています。

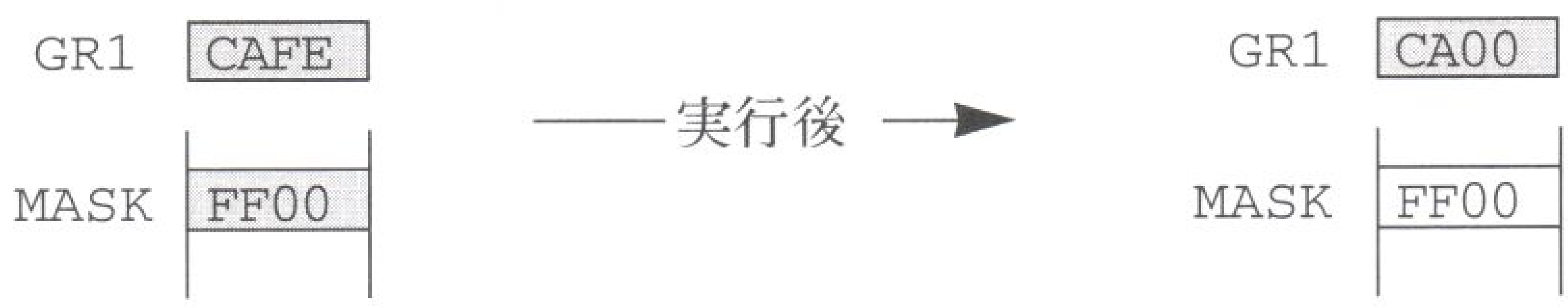
この演算をビットパターンのレベルで見てみましょう。

```
      ABCD   1010101111001101
AND   0F0F   0000111100001111
-----------------------------
      0B0D   0000101100001101  ──→ 0B0D GR5
```

0F0F(16)とANDを取るとF(16)の部分だけが抜き取られます。このようにAND命令では、特定のビットだけを抽出することができます。この処理を**ビット抽出**と呼びます。次の例は領域MASKに設定されたFF00(16)とANDを取って上位8ビットを抽出している例です。

```
AND      GR1,MASK
```

GR1←GR1 AND MASKの領域の値




```
AND      GR1,MSKTBL,GR2
```

GR1←GR1 AND「MSKTBL＋GR2」番地の領域の値



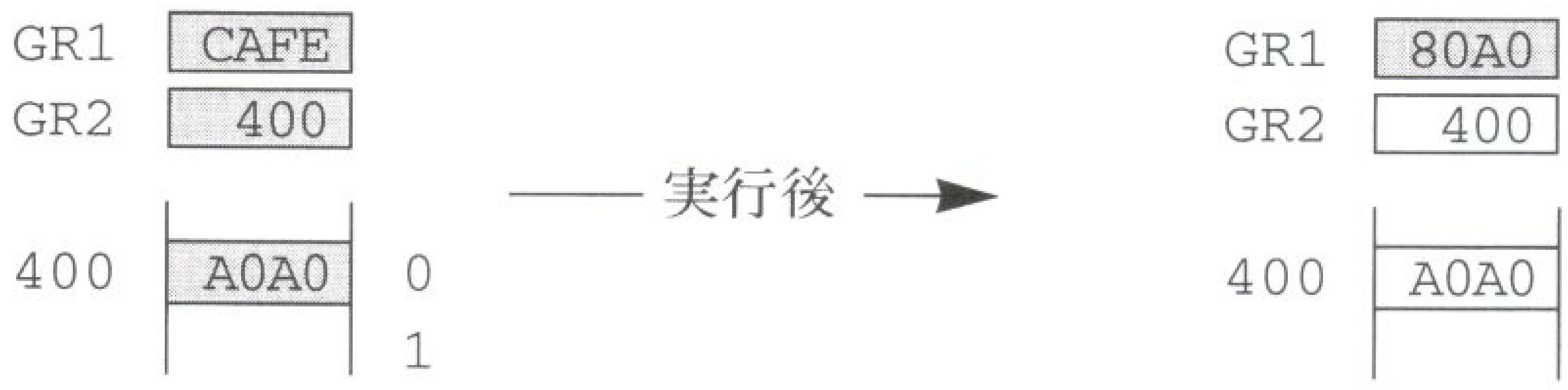

```
AND      GR1,100,GR2
```

GR1←GR1 AND「100番地＋GR2」番地の領域の値(100は絶対番地)



```
AND      GR1,0,GR2
```

GR1←GR1 AND GR2番地の領域の値




CAFE(16)とA0A0(16)の論理積は次のようになります。

```
         CAFE      1100101011111110
   AND   A0A0      1010000010100000
--------------------------------------
         80A0      1000000010100000
```

また、AND命令では、フラグレジスタのSFとZFが、その結果によって設定されます。演算結果がゼロであればZFが1となり、演算結果の最上位ビット(符号ビット)が1であればSFが1となります。

GR1とGR2にさまざまな値を設定したときの演算結果について調べてみましょう。

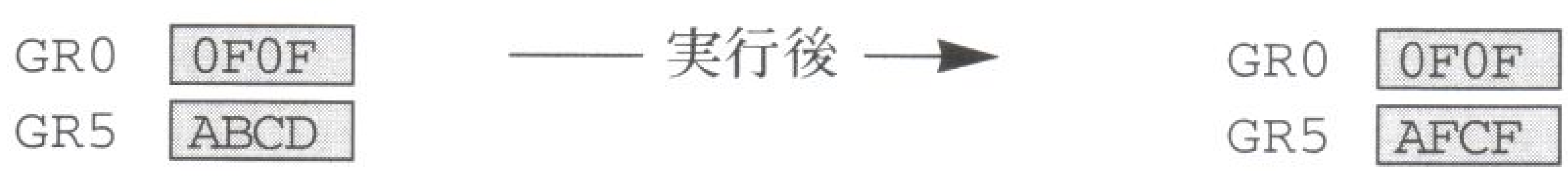
**表6-7　AND命令の計算例とフラグの設定の様子（AND　GR1,GR2）**

| 実行前 | | 実行後 | | 説明 |
|---|---|---|---|---|
| GR1 | GR2 | GR1 | O S Z | |
| FFFF | 0000 | 0000 | - - Z | ゼロクリア：GR2がゼロなので結果は常にゼロになる。ZFが必ず立つ。 |
| FFFF | 0F0F | 0F0F | - - - | ビット抽出：GR1の16進数の3桁目と1桁目を抽出する。 |
| CAFE | EFAC | CAAC | - S - | 論理積：演算結果の最上位ビットが1なのでSFが立つ。 |
| CAFE | 0080 | 0080 | - - - | 特定ビット抽出：0080(16)=0000000010000000(2)でGR1の8ビット目を抽出。ゼロでないのでそこにビットがあることが分かる。 |
| CAFE | 0100 | 0000 | - - Z | 特定ビット抽出：0100(16)=0000000100000000(2)でGR1の9ビット目を抽出。ゼロなのでそこにビットがないことが分かる。 |
| CAFE | 8000 | 8000 | - S - | 最上位ビット抽出：8000(16)=1000000000000000(2)でGR1の最上位ビット抽出。ゼロでないのでそこにビットがあることが分かる。 |
| 7FFF | 8000 | 0000 | - - Z | 最上位ビット抽出：7FFFに対して最上位ビット抽出。ゼロなので、そこにビットがないことが分かる。 |
| BABA | 0001 | 0000 | - - Z | 奇遇判断：GR1の最下位ビットの有無をチェック。ゼロなので、そこにビットがないことがわかる。よってBABAは偶数。 |
| BBBB | 0001 | 0001 | - - - | 奇遇判断：演算結果の最下位ビットが1である。よってBBBBは奇数。 |
| 0068 | 0007 | 0000 | - - Z | 8の倍数の判断：下位3ビットが0なので8の倍数。 |

## 6.3.2　論理和

OR命令は論理和を求めます。

```
OR       GR5,GR0
```



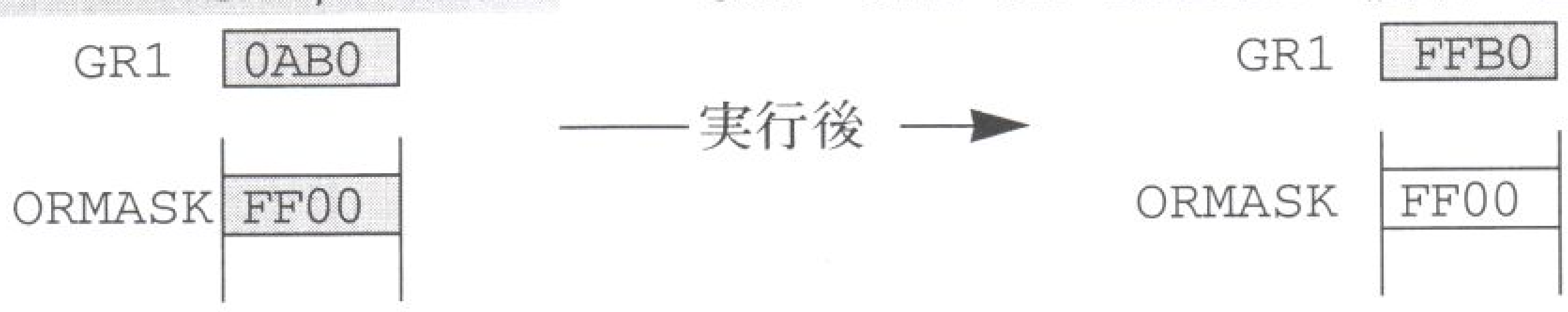

この演算では、ABCD　OR　0F0Fの演算を行い、その結果を再びGR5にセットしています。

この演算をビットパターンのレベルで見てみましょう。

```
        ABCD      1010101111001101
OR      0F0F      0000111100001111
----------------------------------
        AFCF      1010111111001111  ──→ [AFCF] GR5
```

さまざまなパターンのOR命令について調べてみましょう。

`OR　　GR1,ORMASK`　　GR1←GR1 OR ORMASKの領域の値



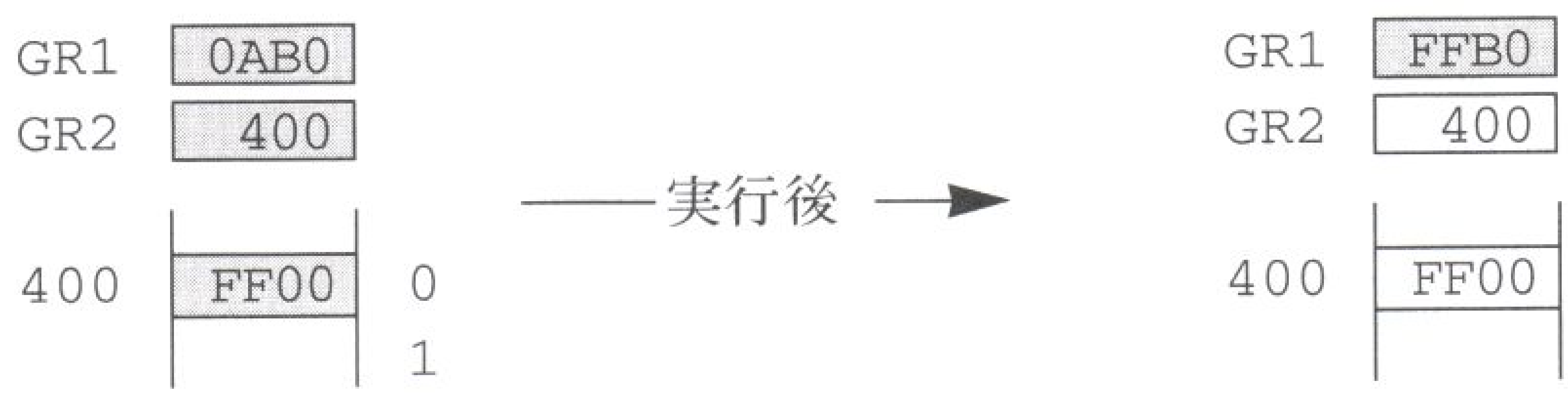

`OR　　GR1,MSKTBL,GR2`　　GR1←GR1 OR「MSKTBL＋GR2」番地の領域の値




`OR　　GR1,100,GR2`　　GR1←GR1 OR「100番地＋GR2」番地の領域の値（100は絶対番地）




`OR　　GR1,0,GR2`　　GR1←GR1 AND GR2番地の領域の値




また、OR命令では、フラグレジスタのSFとZFが、その結果によって設定されます。演算結果がゼロであればZFが1となり、演算結果の最上位ビット(符号ビット)が1であればSFが1となります。

GR1とGR2にさまざまな値を設定したときの演算結果について調べてみましょう。

**表6-8 OR命令の計算例とフラグの設定の様子（OR GR1,GR2）**

| 実行前 | | 実行後 | | 説明 |
|---|---|---|---|---|
| GR1 | GR2 | GR1 | O S Z | |
| 1001 | 0100 | 1101 | - S - | ビット設定：GR1の9ビット目に1を設定。 |
| 0041 | 0020 | 0061 | - - - | ビット設定：大文字コードを小文字コードへ変換(下記参照)。 |
| 7ABC | 8000 | FABC | - S - | ビット設定：GR1の最上位ビットを1にセット。 |

OR命令を用いると、特定のビットを立てる(1にする)ことができます。このように、特定のビットを1にすることを**ビット設定**と呼びます。この演算を用いると、たとえば、大文字コードを小文字コードに変換することができます。まず、いくつかの文字コードを示します。

**表6-9 大文字コードと小文字コードのビットパターン**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | 0100 0001 | B | 0100 0010 | C | 0100 0011 | Z | 0100 1010 |
| a | 0110 0001 | b | 0110 0010 | c | 0110 0011 | z | 0110 1010 |

表6-9に示すように、COMET IIが採用しているJIS X0201(4.4節参照)の文字コードでは英大文字の文字コードの5ビット目を1にすると英小文字の文字コードになります。したがって、大文字のコードと$100000_{(2)}=20_{(16)}$と論理和を取ると、それは、小文字の文字コードになります。

```
       0041    0000 0000 0100 0001  大文字Aの文字コード
    OR 0020    0000 0000 0010 0000
    -----------------------------------
       0051    0000 0000 0110 0001  小文字aの文字コード
```

## 6.3.3 排他的論理和

XOR命令は排他的論理和を求めます。

```
XOR      GR5,GR0
```

```
GR0 [0F0F]   ―― 実行後 ――▶   GR0 [0F0F]
GR5 [ABCD]                   GR5 [A4C2]

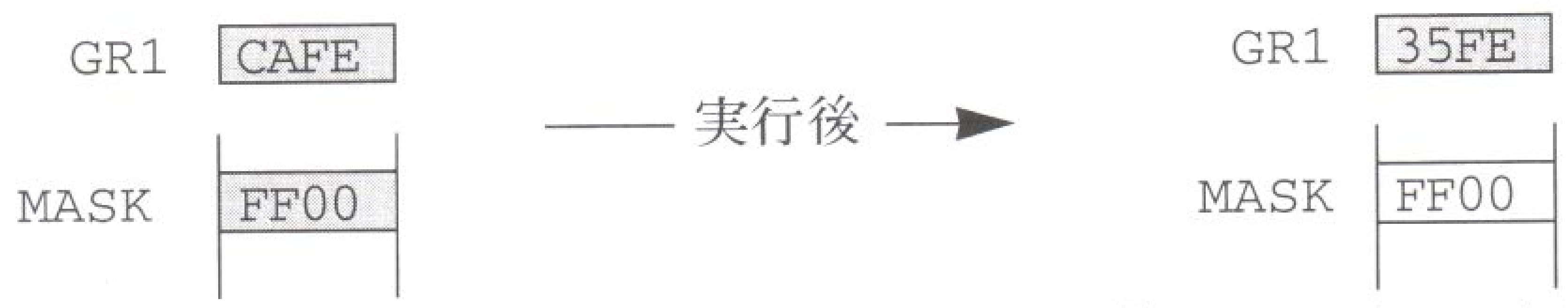
        GR5  ABCD   1010 1011 1100 1101
XOR     GR0  0F0F   0000 1111 0000 1111
--------------------------------------------
             A4C2   1010 0100 1100 0010 ──▶ [A4C2]  GR5
```

この演算では、ABCD　XOR　0F0F の演算を行い、その結果を再びGR5にセットしています。

ところで、ある値と$FFFF_{(16)}$との排他的論理和を取ると、そのビットを逆転させることが出来ます。この処理を**ビット反転**と呼びます。

```
       ABCD  1010 1011 1100 1101
XOR    FFFF  1111 1111 1111 1111
---------------------------------
       5432  0101 0100 0011 0010
```

`XOR GR1,MASK` GR1←GR1 XOR MASKの領域の値



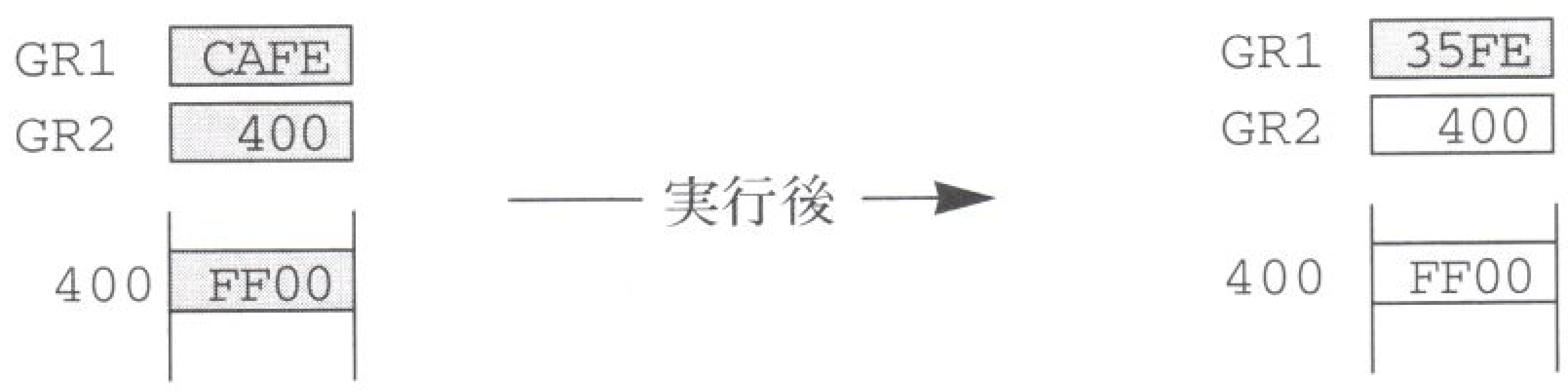

`XOR GR1,MSKTBL,GR2` GR1←GR1 XOR「MSKTBL+GR2」番地の領域の値




`XOR GR1,100,GR2` GR1←GR1 XOR「100番地+GR2」番地の領域の値(100は絶対番地)




`XOR GR1,0,GR2` GR1←GR1 XOR GR2番地の領域の値




また、XOR命令では、フラグレジスタのSFとZFが、その結果によって設定されます。演算結果がゼロであればZFが1となり、演算結果の最上位ビット(符号ビット)が1であればSFが1となります。

GR1とGR2にさまざまな値を設定したときの演算結果について調べてみましょう。

**表6-10 XOR命令の計算例とフラグの設定の様子(XOR GR1,GR2)**

| 実行前 | | 実行後 | | 説明 |
|---|---|---|---|---|
| GR1 | GR2 | GR1 | O S Z | |
| 1234 | FFFF | EDCB | - S - | ビット反転:FFFFと排他的論理和を取るとビットが反転する。 |
| 1234 | 0F0F | 1D3B | - - - | ビット反転:特定のビットだけ反転させる。 |
| 1234 | 0001 | 1235 | - - - | 奇遇反転:最下位ビットを反転させて、奇遇を逆にする。 |
| 1235 | 0001 | 1234 | - - - | 奇遇反転:最下位ビットを反転させて、奇遇を逆にする。 |
| 1234 | 1234 | 0000 | - Z - | ゼロクリア:同じ値の排他的論理和を取ると0になる。XOR GR1,GR1でGR1を0にできる。 |

## 6.4 比較演算命令

**比較演算命令**は、2つのデータの大小を比較し、その結果をフラグレジスタに設定する演算です。主記憶装置と汎用レジスタの間のデータの比較演算、あるいは汎用レジスタ間のデータの比較演算があります。この命令は単独で使われることはなく、この比較演算の次に6.6節で学習する分岐命令と組み合わせて使います。

比較演算命令には、符号付きデータを対象とする**算術比較**(CPA命令)と符号無しデータを対象とする**論理比較**(CPL命令)の 2 つがあります。

**表6-11 比較演算命令**

<table>
<tr><th rowspan="2">命　　令</th><th colspan="2">書 き 方</th><th rowspan="2">命令の説明</th><th rowspan="2">FRの設定</th></tr>
<tr><th>命令コード</th><th>オペランド</th></tr>
<tr><td>算術比較<br>ComPare Arithmetic</td><td>CPA</td><td>r1,r2<br>r,adr[,x]</td><td rowspan="2">(r1)と(r2)、又は(r)と(実効アドレス)の算術比較又は論理比較を行い、比較結果によって、FRに次の値を設定する。<br><table><tr><th rowspan="2">比較結果</th><th colspan="2">FRの値</th></tr><tr><th>SF</th><th>ZF</th></tr><tr><td>(r1)＞(r2)<br>(r)＞(実効アドレス)</td><td>0</td><td>0</td></tr><tr><td>(r1)＝(r2)<br>(r)＝(実効アドレス)</td><td>0</td><td>1</td></tr><tr><td>(r1)＜(r2)<br>(r)＜(実効アドレス)</td><td>1</td><td>0</td></tr></table></td><td rowspan="2">OFは常に0<br>SF, ZFは設定される</td></tr>
<tr><td>論理比較<br>ComPare Logical</td><td>CPL</td><td>r1,r2<br>r,adr[,x]</td></tr>
</table>

たとえば、算術比較「CPA　GR1,GR2」は、レジスタGR1とレジスタGR2の大小関係をチェックして、表6-12に従ってフラグレジスタを設定します。なお、表6-12に示すようにCPA命令ではOFは1になることはなく、常に0になります。

**表6-12 算術比較「CPA GR1, GR2」実行時のフラグレジスタの設定**

| | 1になるとき | 0になるとき |
|---|---|---|
| OF(Overflow Flag) | ありえない | 常に0 |
| SF(Sign Flag) | GR1 < GR2 | GR1 ≧ GR2 |
| ZF(Zero Flag) | GR1 = GR2 | GR1 ≠ GR2 |

一方、論理比較「CPL　GR1,GR2」のときに、演算結果とフラグレジスタの関係は次のようになります。なお、表6-13に示すようにCPL命令ではOFは1になることはなく、常に0になります。

**表6-13 論理比較「CPL GR1, GR2」実行時のフラグレジスタの設定**

| | 1になるとき | 0になるとき |
|---|---|---|
| OF(Overflow Flag) | ありえない | 常に0 |
| SF(Sign Flag) | GR1 < GR2 | GR1 ≧ GR2 |
| ZF(Zero Flag) | GR1 = GR2 | GR1 ≠ GR2 |

算術比較(CPA)の表6-12と論理比較(CPL)の表6-13には違いが無いように見えますが、表6-14に示すようにフラグレジスタに設定される値が異なります。

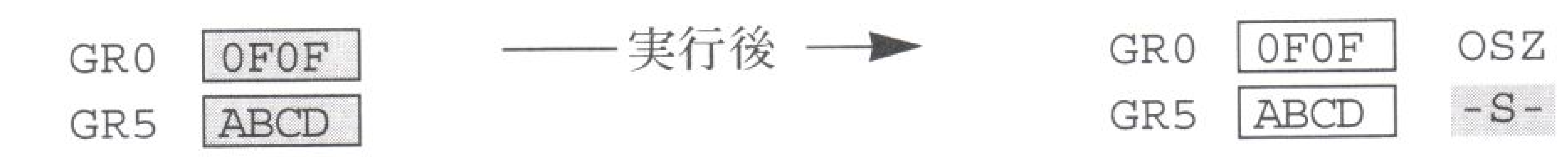
**表6-14 さまざまなデータの算術比較、論理比較を行ったときの表**

| 16進数 | | 算術比較 10進数 | | | 論理比較 10進数 | | |
|---|---|---|---|---|---|---|---|
| GR1 | GR2 | GR1 | GR2 | O S Z | GR1 | GR2 | O S Z |
| 0001 | 0001 | 1 | 1 | - - Z | 1 | 1 | - - Z |
| 0001 | FFFF | 1 | -1 | - - - | 1 | 65535 | - S - |
| CCCC | DDDD | -13108 | -8739 | - S - | 52428 | 56797 | - S - |
| 7FFF | 00FF | 32767 | 255 | - - - | 32767 | 255 | - - - |
| 8100 | 0FFF | -32512 | 4095 | - S - | 33024 | 4095 | - - - |
| 0ABC | 0DEF | 2748 | 3567 | - S - | 2748 | 3567 | - S - |
| CCCC | CCCC | -13108 | -13108 | - - Z | 52428 | 52428 | - - Z |

## 6.4.1 CPA命令

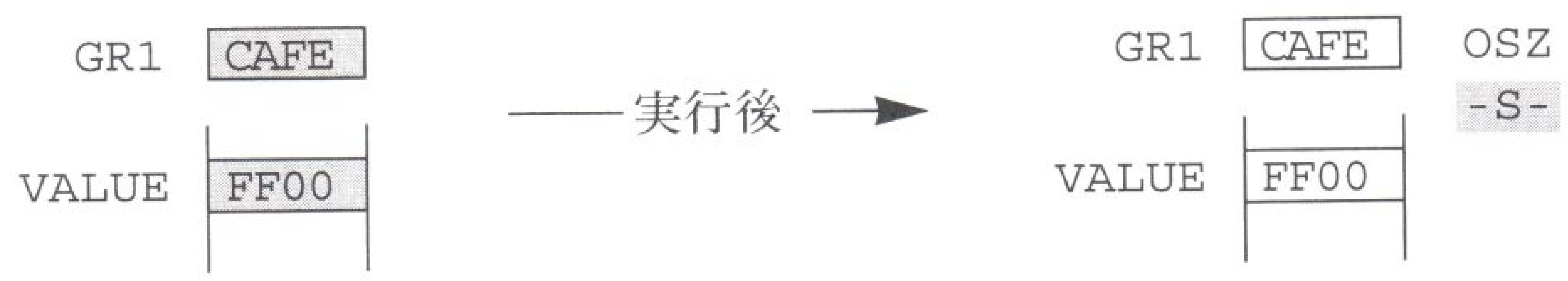
CPA命令について例を示しながら説明します。

```
CPA      GR5,GR0
```




上記の例では、符号付きで、ABCD(16)＝－21555, 0F0F(16)＝3855となりますから、GR5<GR0となってSFに1が立ちます。

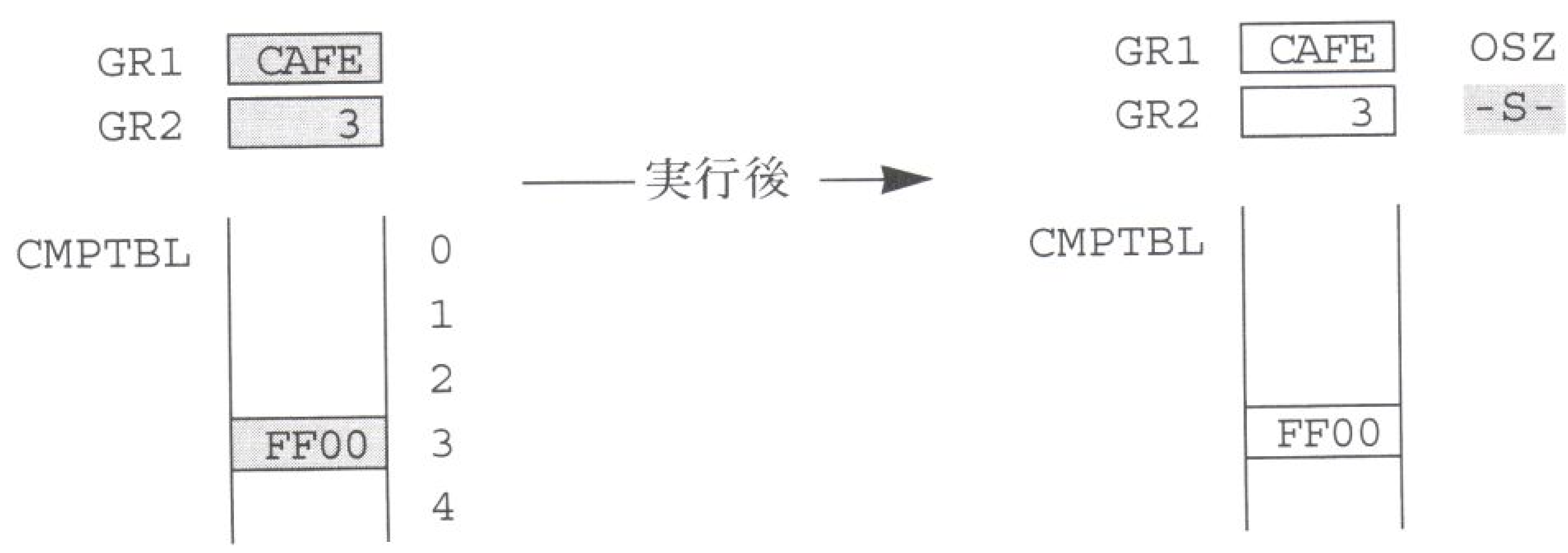
```
CPA      GR1,VALUE
```
GR1とVALUEの領域の値を算術比較する



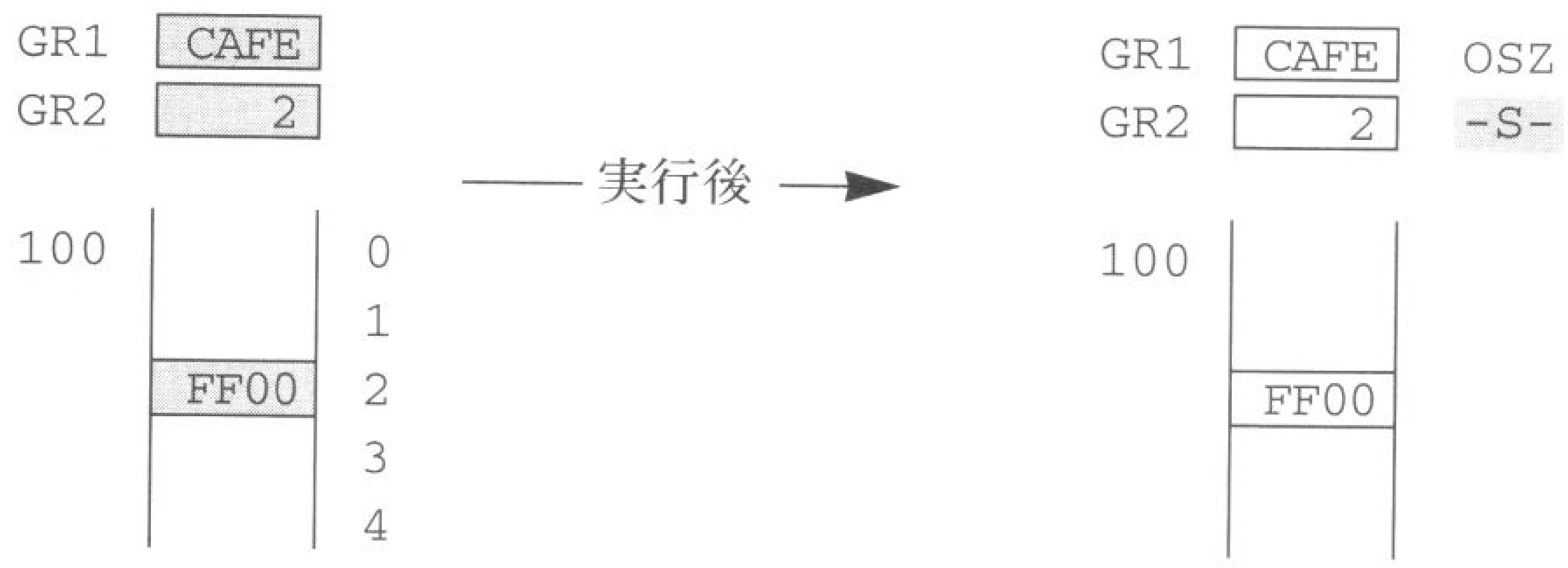

上記の例では、符号付きでCAFE(16)=－13570、FF00(16)=－256ですから、「GR1の値 < VALUEの値」となってSFに1が立ちます。

以下に、これ以外の例を示します。

```
CPA      GR1,CMPTBL,GR2
```
GR1と「CMPTBL＋GR2」番地の領域の値を算術比較する



`CPA GR1,100,GR2` GR1と「100番地＋GR2」番地の領域の値を算術比較する（100は絶対番地）




`CPA GR1,0,GR2` GR1とGR2番地の領域の値を算術比較する




### 6.4.2 CPL命令

`CPL GR5,GR0`




上記の例では、符号無しで、ABCD(16)＝43981, 0F0F(16)＝3855となりますから、GR5＞GR0となってSFもZFも設定されません。

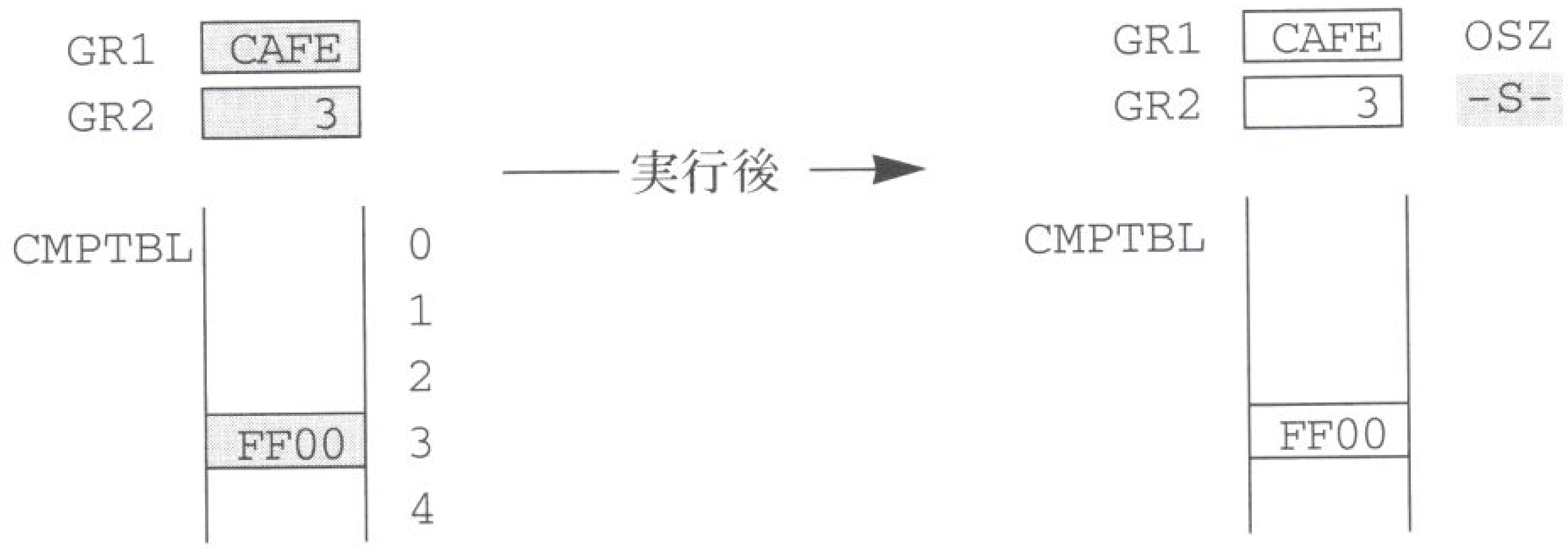
`CPL GR1,VALUE` GR1とVALUEの領域の値を比較する



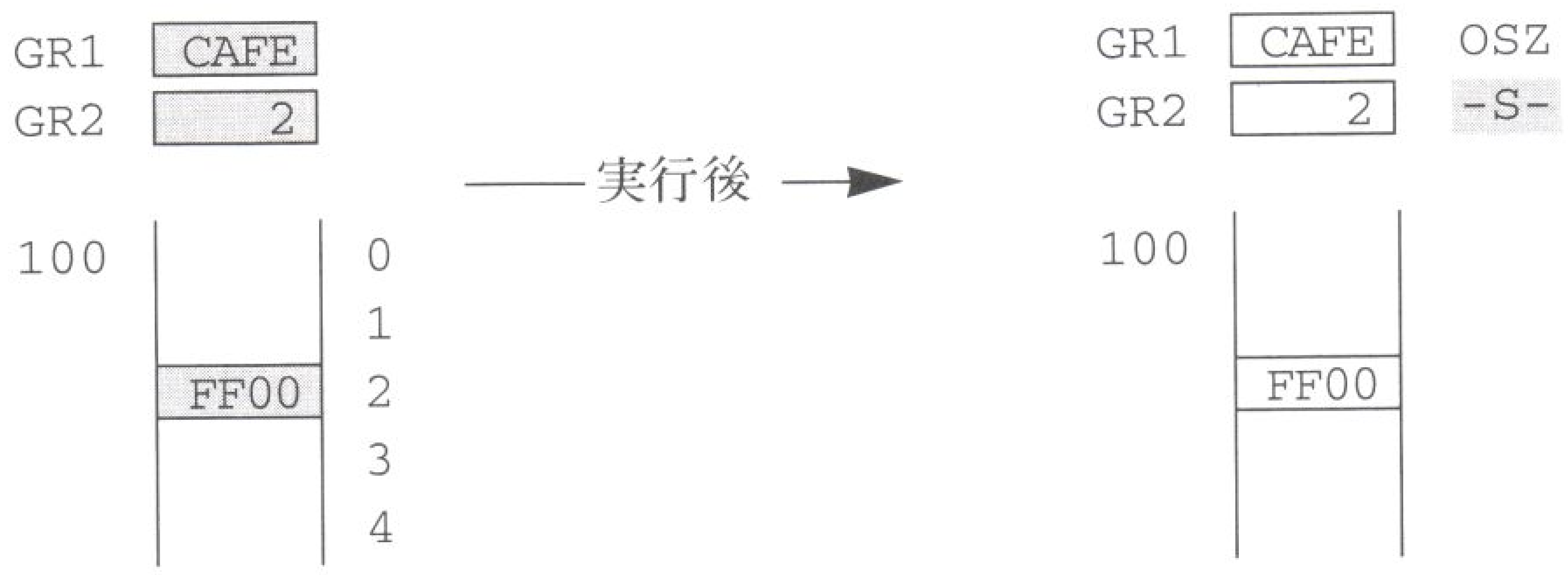

上記の例では、符号無しで、CAFE(16)＝51966, FF00(16)＝65280ですから、「GR1＜VALUEの値」となってSFに1が立ちます。

以下に、これ以外の例を示します。

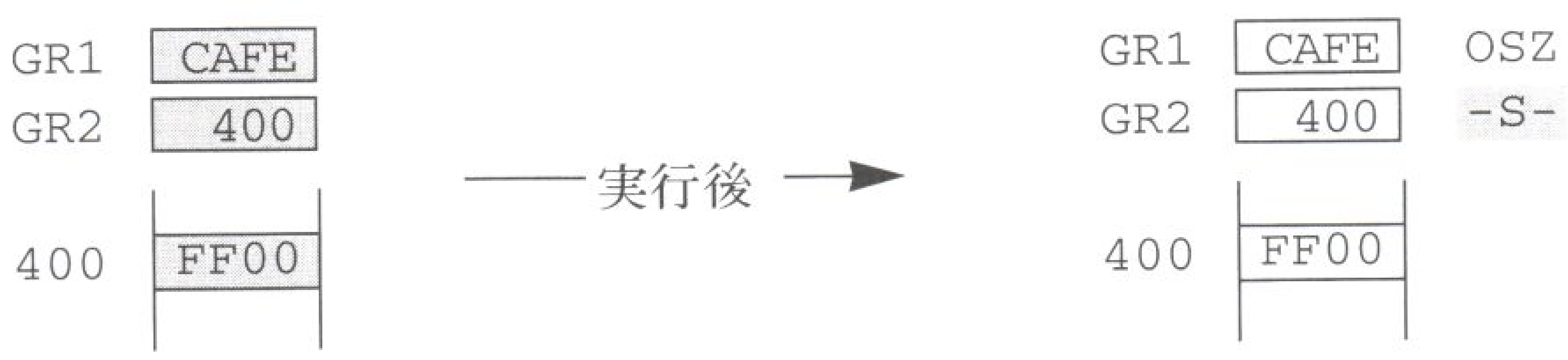
`CPL GR1,CMPTBL,GR2` GR1と「CMPTBL+GR2」番地の領域の値を論理比較する




`CPL GR1,100,GR2` GR1と「100番地+GR2」番地の領域の値を論理比較する(100は絶対番地)




`CPL GR1,0,GR2` GR1とGR2番地の領域の値を論理比較する



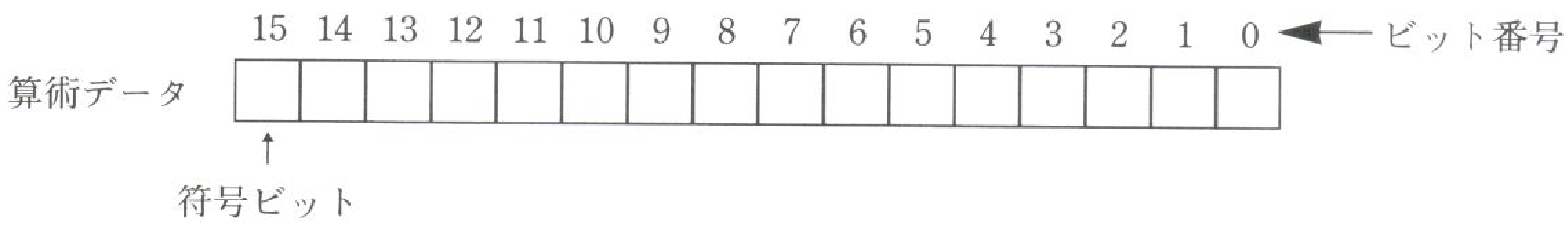
# 6.5 シフト演算命令

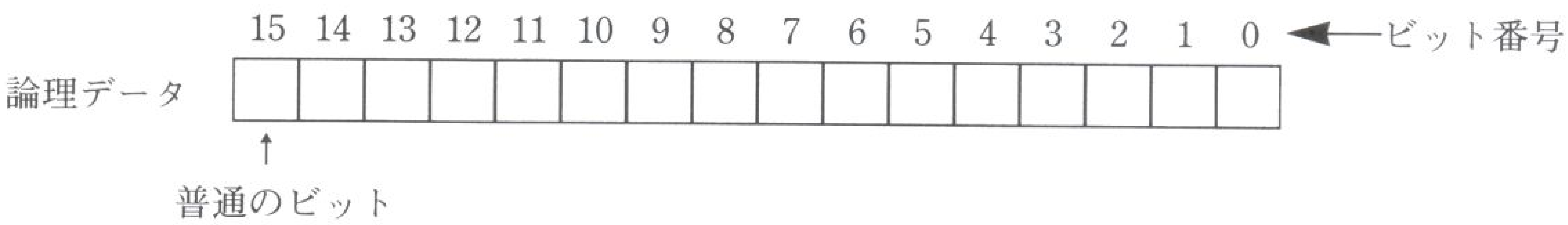
シフト演算命令は、汎用レジスタのデータをシフトするための命令です。

表6-15 シフト演算命令

| 命令 | 書き方 | | 命令の説明 | FRの設定 |
|---|---|---|---|---|
| | 命令コード | オペランド | | |
| 算術左シフト<br>Shift Left Arithmetic | SLA | r,adr[,x] | 実効アドレスで指定されたビット数だけ左に算術シフトする | OF, SF, ZFが設定される |
| 算術右シフト<br>Shift Right Arithmetic | SRA | r,adr[,x] | 実効アドレスで指定されたビット数だけ右に算術シフトする | OF, SF, ZFが設定される |
| 論理左シフト<br>Shift Left Logical | SLL | r,adr[,x] | 実効アドレスで指定されたビット数だけ左に論理シフトする | OF, SF, ZFが設定される |
| 論理右シフト<br>Shift Right Logical | SRL | r,adr[,x] | 実効アドレスで指定されたビット数だけ右に論理シフトする | OF, SF, ZFが設定される |

表6-15を見ると、算術シフトと論理シフトの2種類の命令があることがわかります。この2つの違いは、符号をどのように処理するかということです。算術データ(いわゆる符号付きデータ)と論理データ(いわゆる符号無しデータ)の違いは最上位ビットの扱いにあります。

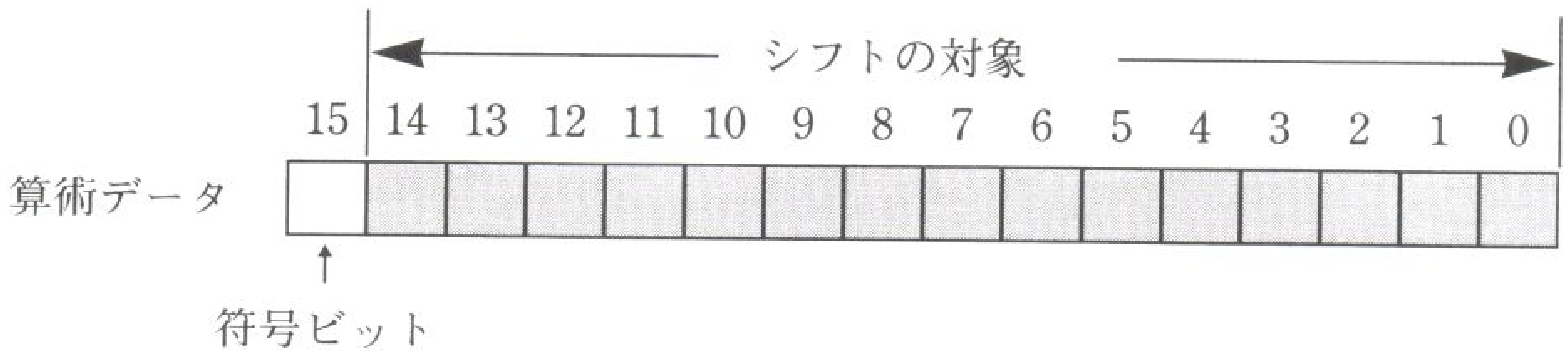



算術データでは、最上位ビットは符号ビットなので特別扱いになります。それぞれの違いは、個々の命令の説明の中で行いましょう。

## 6.5.1 SLA命令

SLA命令は、実効アドレスで指定された数だけ、汎用レジスタの内容(値)を左に算術シフトします。算術シフトでは、第15ビット目(いわゆる符号ビット)がシフトの対象にならず前の値がそのまま残ります(この理由は後述します)。



まず、SLA命令の構文について見ていくことにしましょう。

| | |
|---|---|
| `SLA GR1,3` | 実効アドレスは3ですから、GR1の値を3ビット算術左シフトします。 |
| `SLA GR0,0,GR1` | 実効アドレスはGR1の値ですから、GR0の値をGR1の値だけ算術左シフトします。 |
| `SLA GR2,2,GR3` | 実効アドレスは「2+GR3の値」ですから、GR2の値を「2+GR3の値」だけ算術左シフトします。 |

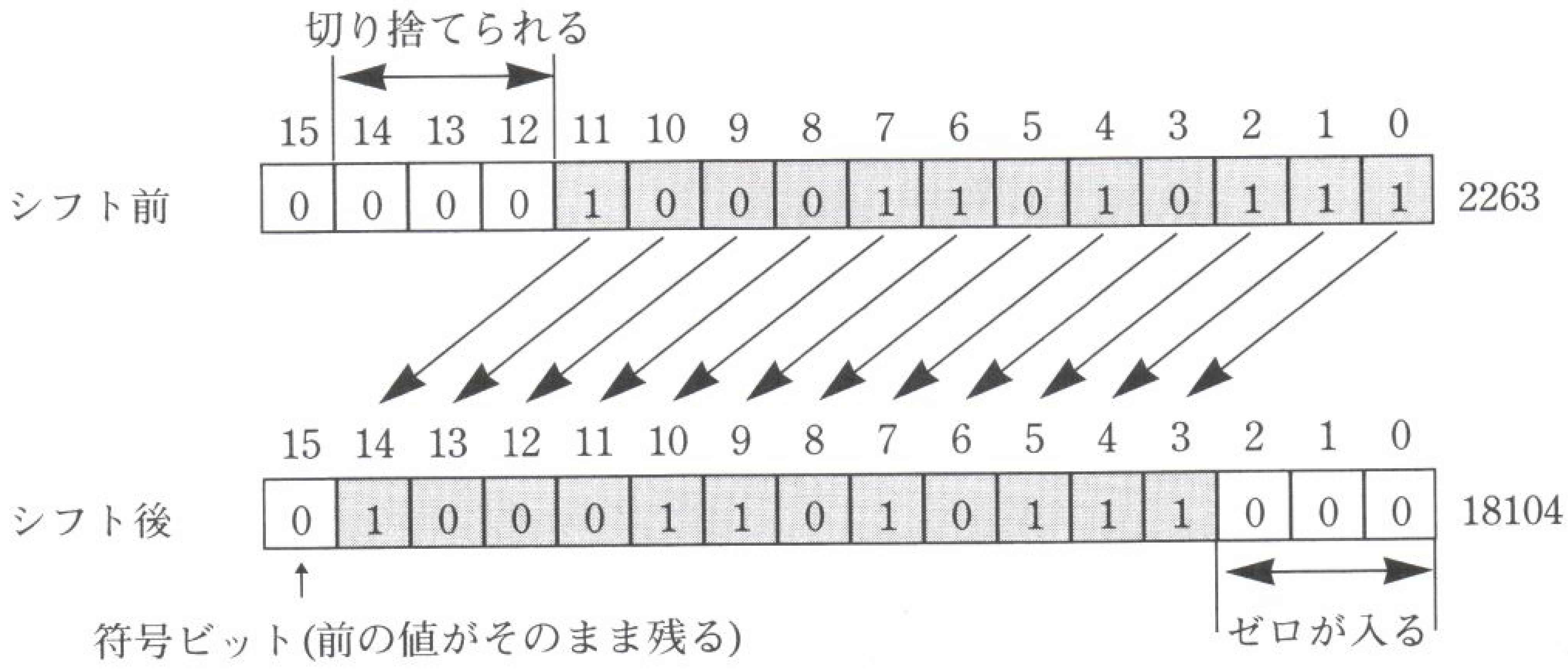
この他に構文としては、次の例のようにラベルを使う方法がありますが、実際のプログラムの中ではほとんど使うことはないでしょう。

| | |
|---|---|
| `SLA GR2,LBL,GR1` | 実効アドレスは「LBLのラベルのアドレス＋GR1の値」となり、この分だけ算術左シフトします。 |

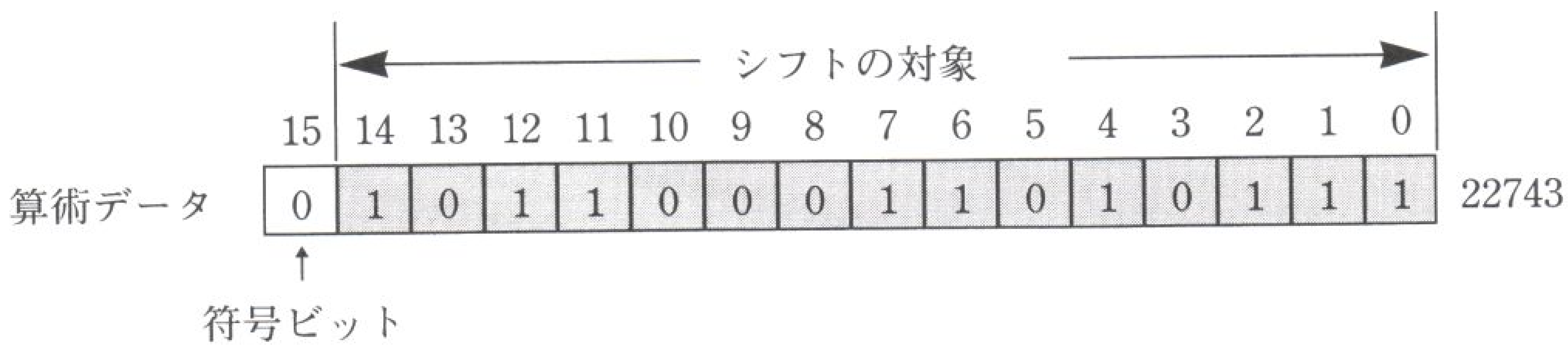
さて、今、次のデータがレジスタGR1にあるとします。



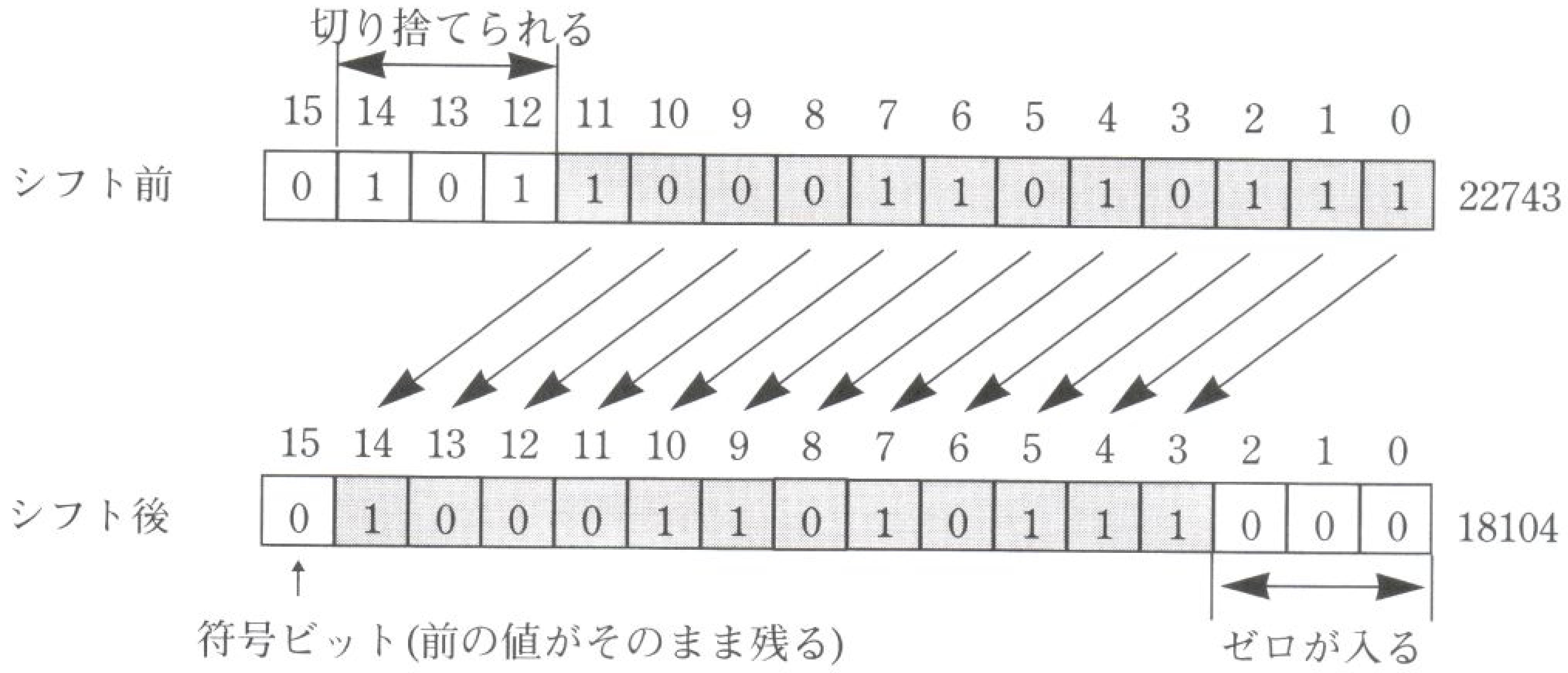

これを「SLA　GR1,3」で3ビット左に算術シフトします。すると次のようになります。




ちょうど、18104＝2263×8ですから、元の数の$2^3$＝8倍になっています。
それでは、今度は、次の数を3ビット左に算術シフトしてみましょう。



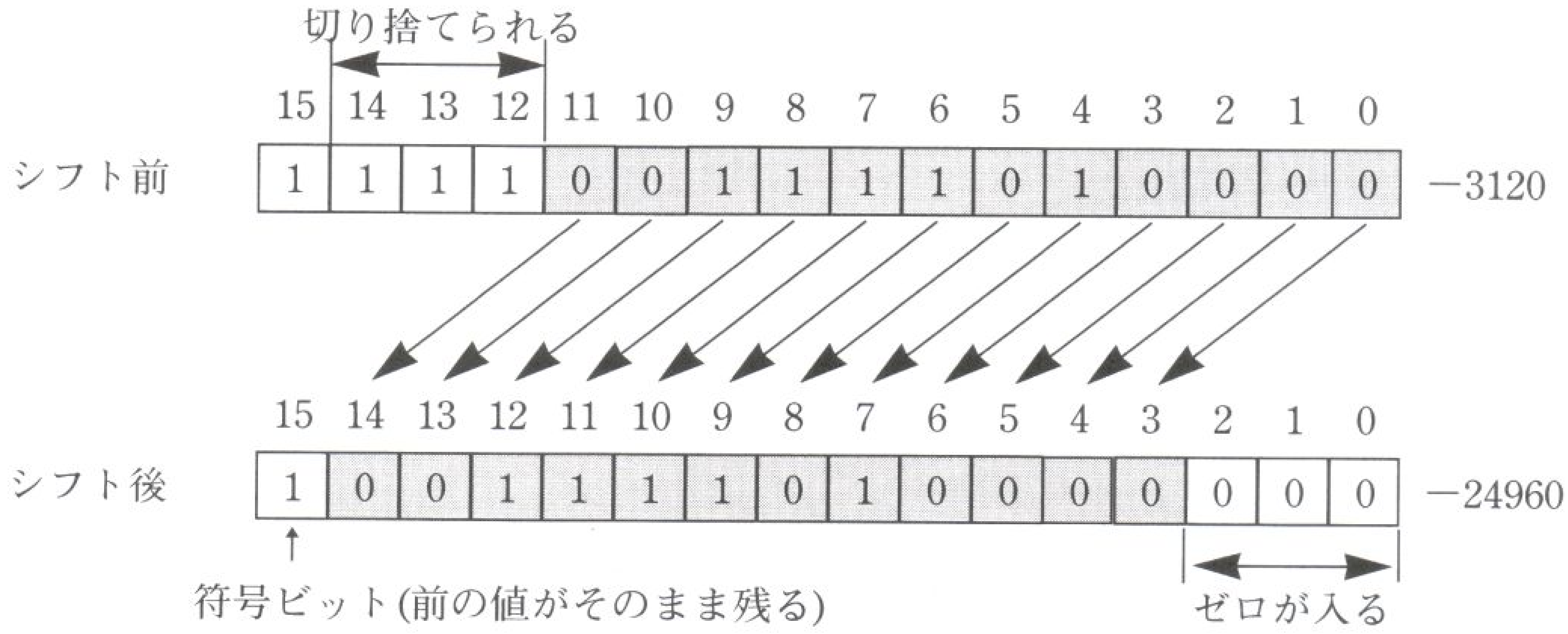
すると、次のようになります。



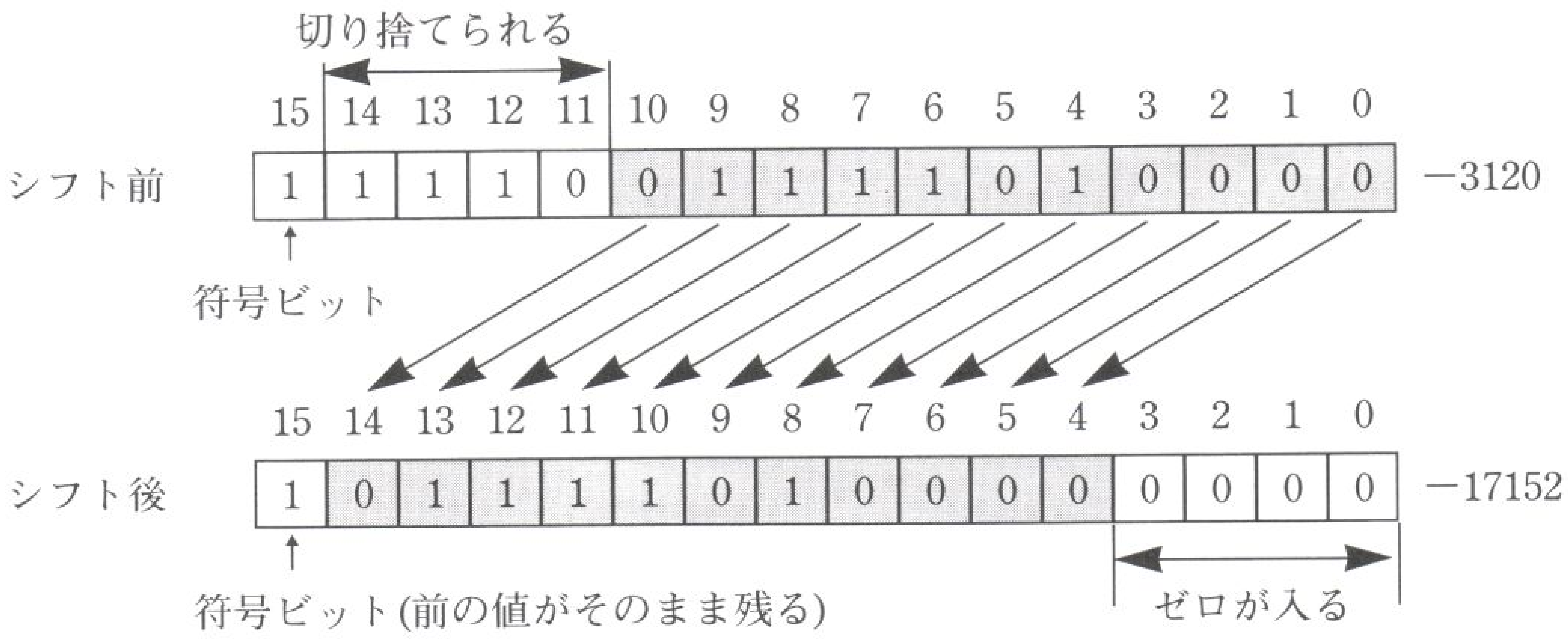

この場合には、上位3ビットがゼロではないので、その部分が切り捨てられるため、22743×8＝181944にはなりません。

さて、データを単なるビットパターンとみなせば、上記の例では12ビット目の1が15ビット目にシフトされるべきなのですが、算術左シフトの場合には、15ビット目は符号ビットなので特別扱いとなり、前の値がそのまま残ります。

それでは、負の数の算術左シフトはどのように処理されるのでしょうか。今度は、−3120を3ビット左に算術シフトしてみましょう。

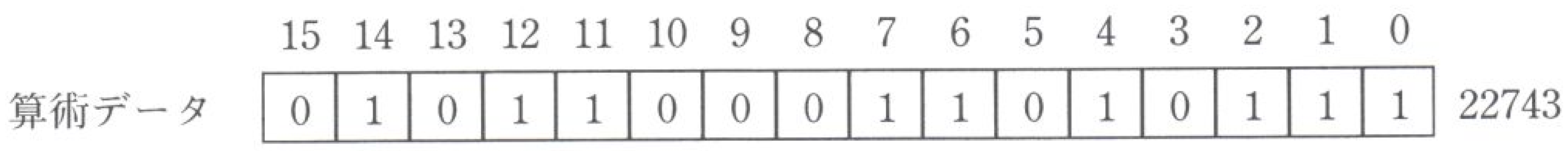



さて、−3120×8＝−24960となり、上位3ビットの1が切り捨てられても符号ビットに1を設定しているので、ちゃんと8倍になっていることが分かると思います。もし、符号ビットを1にしないと、0001111010000000(2)となりこの値は7808ですから、全く異なる値になってしまいます。このように符号ビットを残すことによって、シフト演算の結果が、−32768～32767の間にある限りは、$2^n$倍されることになります。ただし、この範囲を越えてしまう場合には、正しい答えが格納されませんので注意が必要です。たとえば、−3120×16の場合、−3120×16＝−49920となって、その範囲（−32768～32767）に入りません。16倍にするには4ビット左に算術シフトします。この様子を以下に図示します。

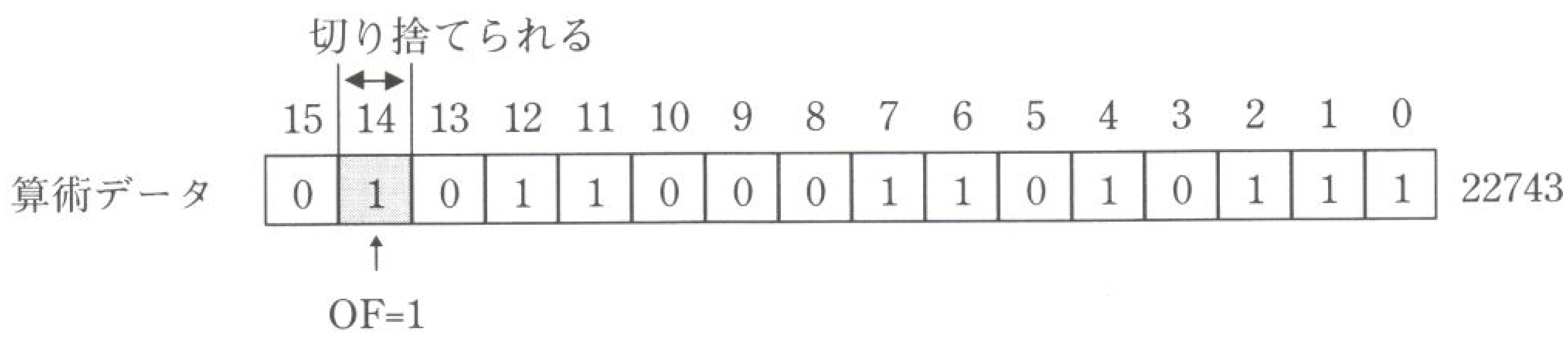



この場合、−17152という数になってしまいます。

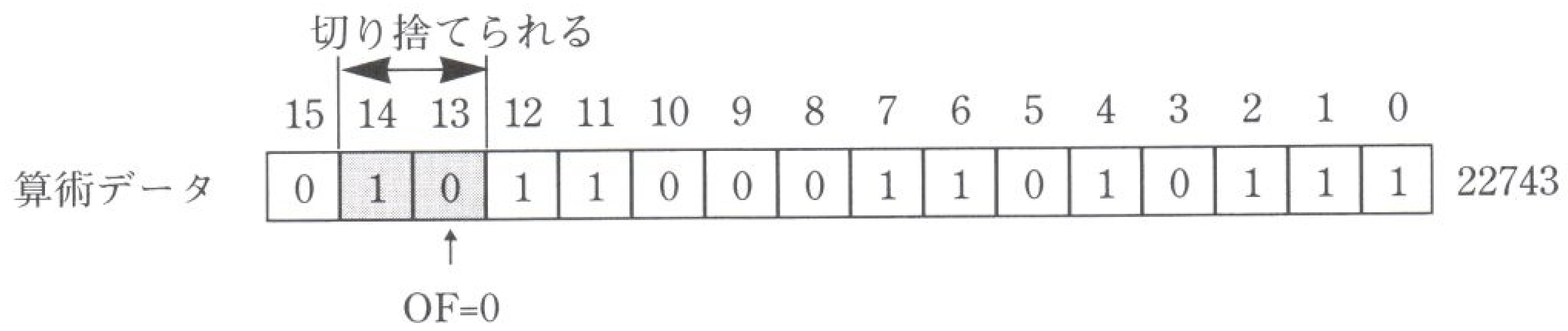
算術左シフト演算時のOF（オーバーフローフラグ）は、ちょっと分かりにくいので、ここで詳しく説明します。まず、22743という数値を1ビットおよび2ビット、−3120という数値を算術左シフトしたときのオーバーフローフラグの設定について考えましょう。



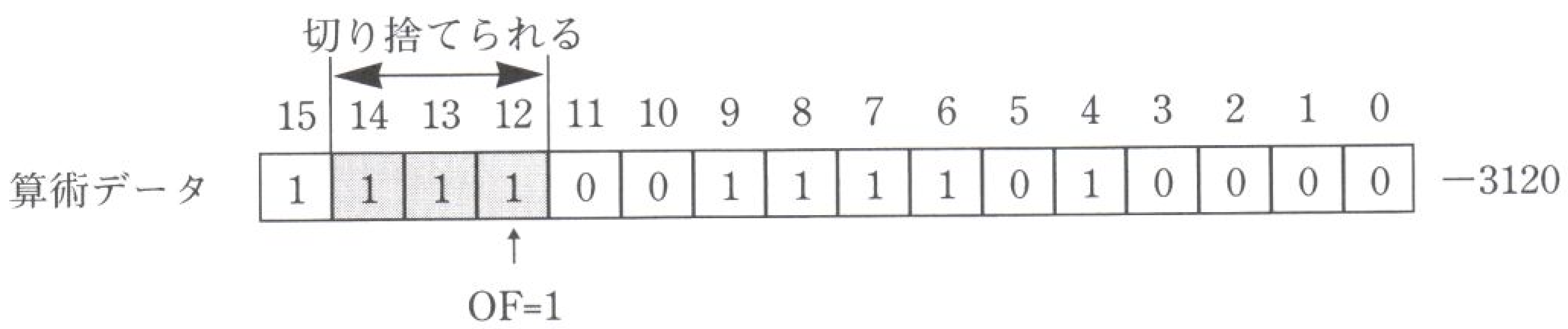

「仕様」では、「一番最後に押し出されたビットがOFにセットされる」と記述されています。22743を左に1ビット算術シフトすると、押し出されるのは14ビット目です。これがOFに入りますのでこの場合、OF＝1となります。




22743を左に2ビット算術シフトしたときに最後に押し出されるのは、13ビット目ですから、この場合、OF＝0となります。

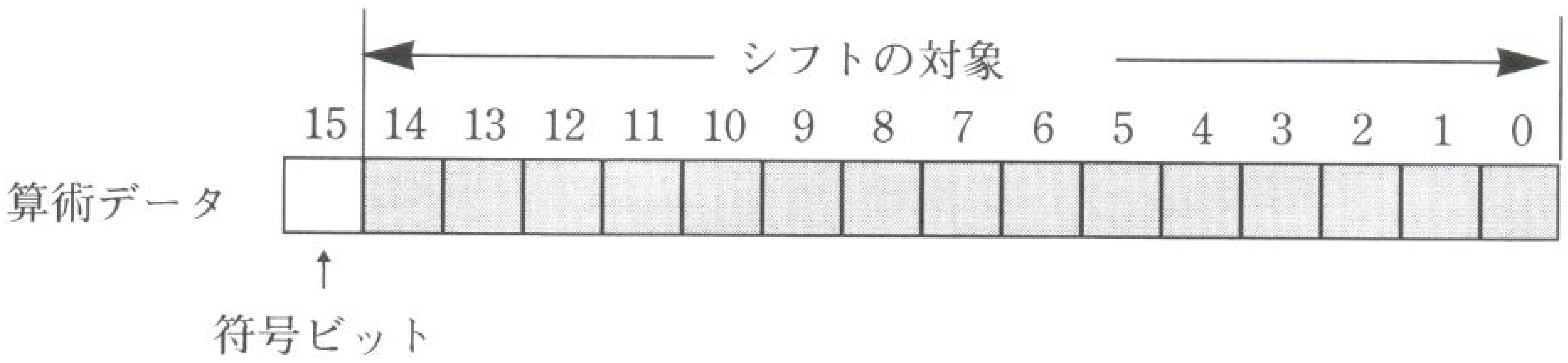



この規則は負の数でも同じです。−3120を左に3ビット算術シフトしたときに、最後に押し出されるビットは12ビット目ですから、その値がOFにセットされます。



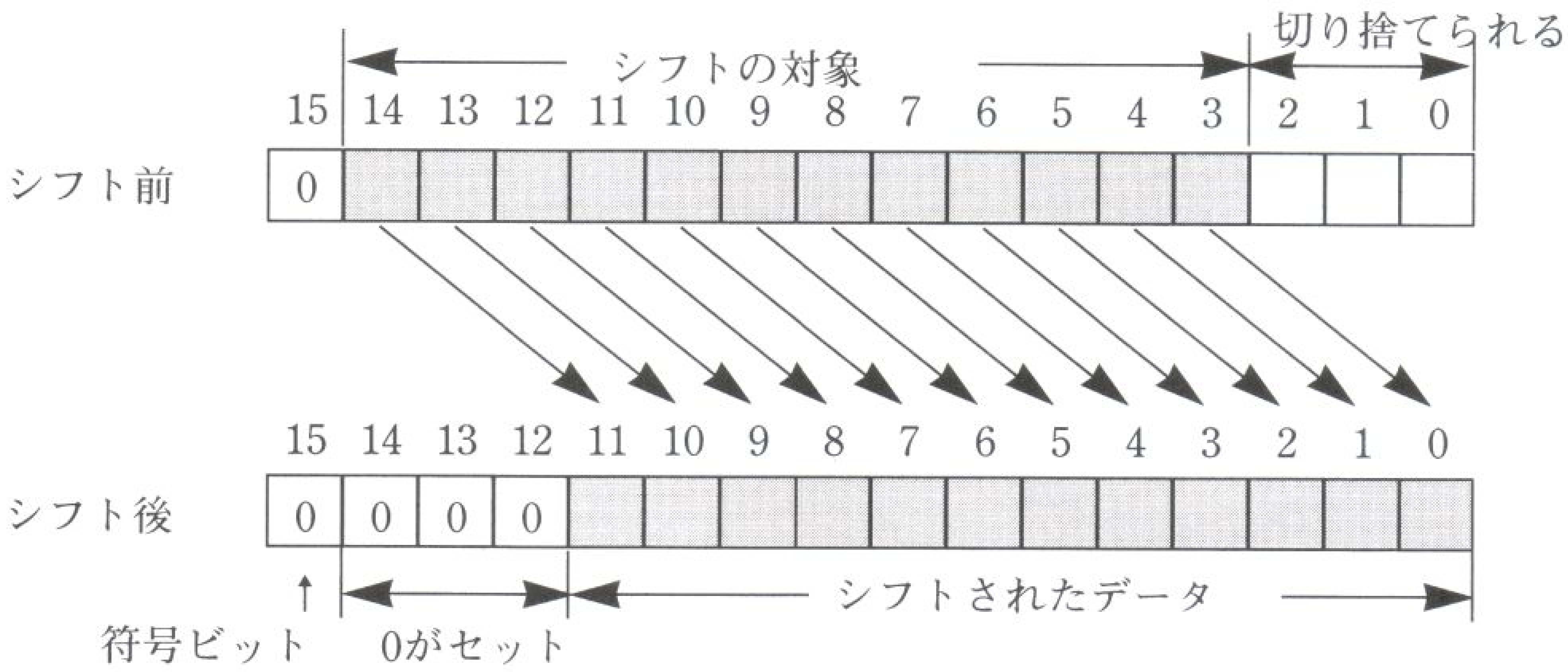
### 6.5.2 SRA命令

SRA命令は、実効アドレスで指定された数だけ、汎用レジスタの内容(値)を右に算術シフトします。前述のように、算術シフトでは、第15ビット目(いわゆる符号ビット)がシフトの対象にならず前の値がそのまま残ります。

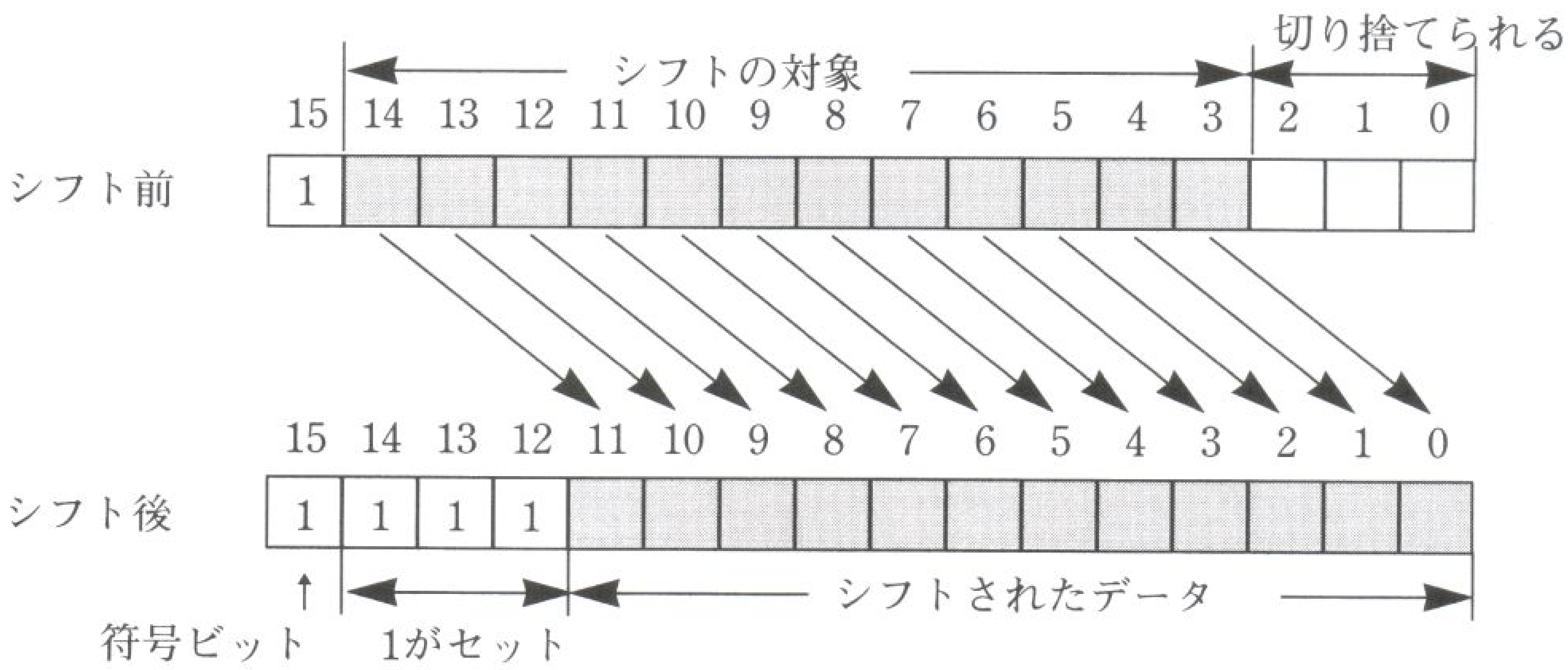



また、シフトされて空いた上位ビットには符号ビットと同じビットがそのまま入ります。それでは、3ビット右に算術シフトすることを考えてみましょう。

正の場合、符号ビットには0が入っていますから次のようになります。




負の場合、符号ビットには1が入っていますから次のようになります.



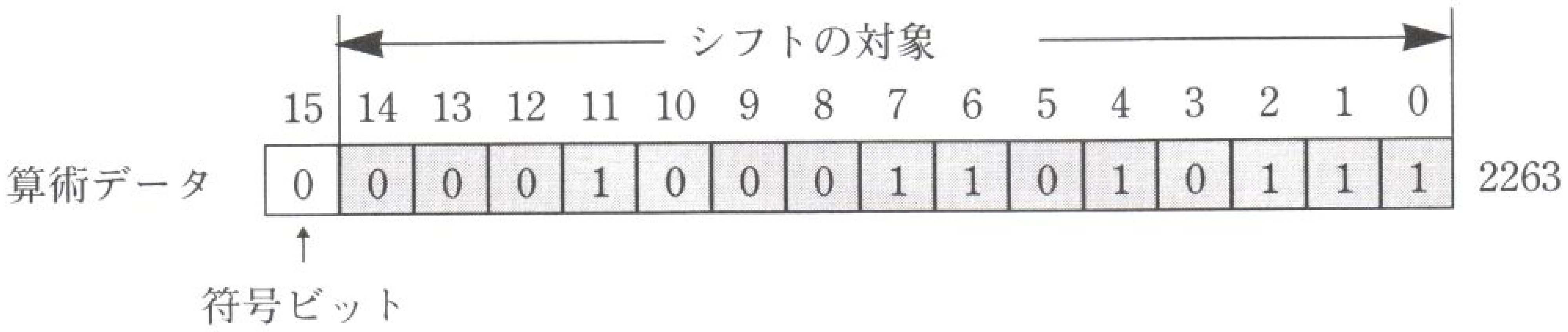

それでは、まず、SRA命令の構文について見ていくことにしましょう。

| | | |
|---|---|---|
| `SRA` | `GR1,3` | 実効アドレスは3ですから、GR1の値を3ビット算術右シフトします。 |
| `SRA` | `GR0,0,GR1` | 実効アドレスはGR1の値ですから、GR0の値をGR1の値だけ算術右シフトします。 |
| `SRA` | `GR2,5,GR3` | 実効アドレスは「5+GR3の値」ですから、GR2の値を「5+GR3の値」だけ算術右シフトします。 |

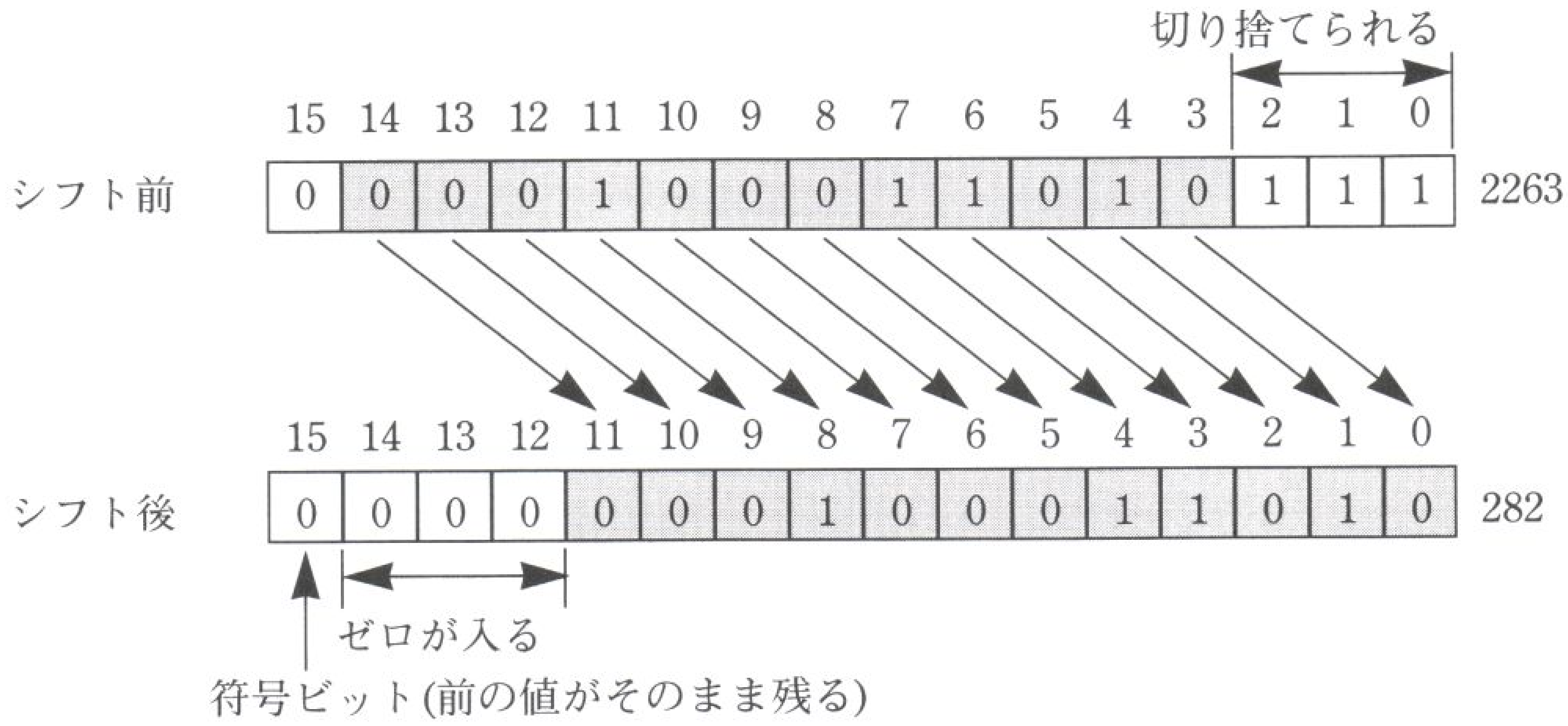
この他に構文としては、次の例のようにラベルを使う方法がありますが、実際のプログラムの中ではほとんど使うことはないでしょう。

| | |
|---|---|
| `SRA    GR2,LBL,GR1` | 実効アドレスは「LBLのラベルのアドレス+GR1の値」となり、この分だけ算術右シフトします。 |

さて、今、次のデータがレジスタGR1にあるとします。

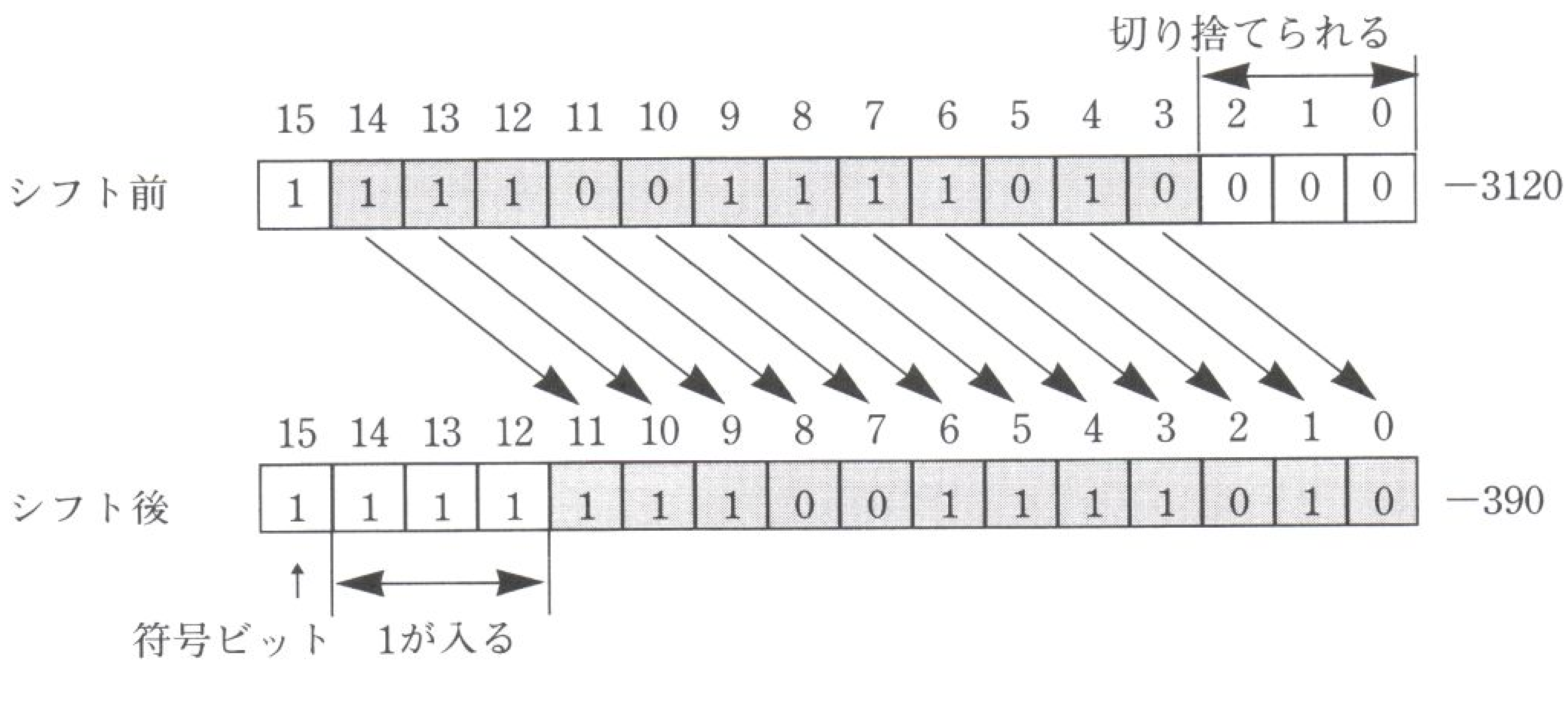



これを「SRA　GR1,3」で3ビット右に算術シフトします。すると次のようになります。




ちょうど、2263/8＝282.875ですから、整数の部分は元の数を$2^3$＝8で割った値になっています。小数点以下(0.875)が残っているのは切り捨てられた部分がゼロでないからです。切り捨てられた部分$111_{(2)}$＝7は、2263を8で割った余りになっています。

それでは、負の数の算術右シフトについて見てみましょう。今度は、次の数を3ビット右に算術シフトしてみましょう。




さて、−3120/8＝−390となり、下位3ビットが切り捨てられても、符号ビット以降に1を設定しているので、ちゃんと8で割った値になっていることが分かると思います。もし、符号ビット以降の3ビ

ットを1にしないと、$1000111001111010_{(2)}$となりこの値は－29062ですから、全く異なる値になってしまいます。このように符号ビットと同じビットで空いた部分を埋めること(これを**符号拡張**と呼びます)によって、算術右シフト演算の結果は、$2^n$で割った値になります。

算術右シフト演算時のOF（オーバーフローフラグ）は、ちょっと分かりにくいので、ここで詳しく説明します。まず、22743という数値を1ビット、4ビット、－3120という数値を3ビット算術右シフトしたときのOFの設定について考えましょう。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 算術データ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 22743 |

「仕様」では、「一番最後に押し出されたビットがOFにセットされる」と記述されています。22743を右に1ビット算術シフトすると、押し出されるのは0ビット目です。これがOFに入りますのでこの場合、OF＝1となります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 算術データ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 22743 |
| | | | | | | | | | | | | | | | | ↑ OF=1 | |

22743を左に4ビット算術シフトしたときに最後に押し出されるのは、3ビット目ですから、この場合、OF＝0となります。

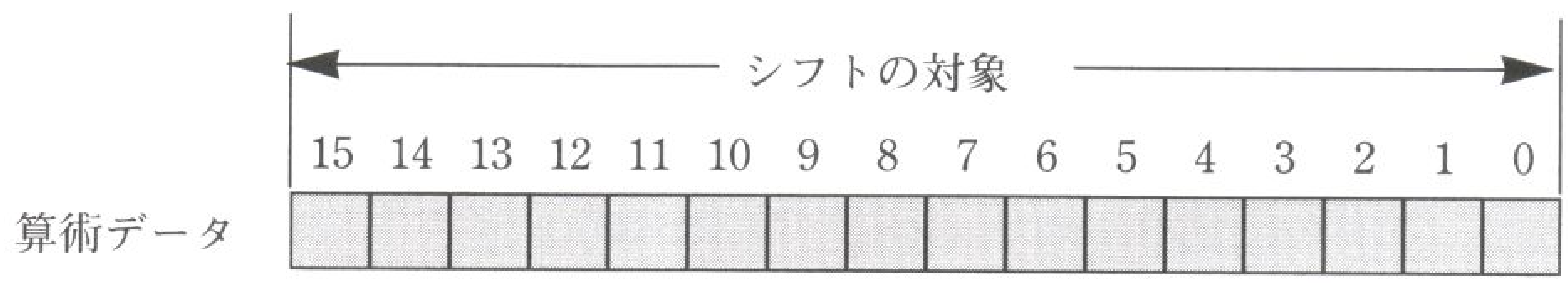
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 算術データ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 22743 |
| | | | | | | | | | | | | | ↑ OF=0 | | | | |

この規則は負の数でも同じです。－3120を右に3ビット算術シフトしたときに最後に押し出されるビットは2ビット目ですから、その値がOFにセットされます。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 算術データ | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | －3120 |
| | | | | | | | | | | | | | | ↑ OF=0 | | | |

### 6.5.3 SLL命令

SLL命令は、実効アドレスで指定された数だけ、汎用レジスタの内容(値)を左に論理シフトします。論理シフトでは、すべてのビットが平等にシフトの対象となります。




まず、SLL命令の構文について見ていくことにしましょう。

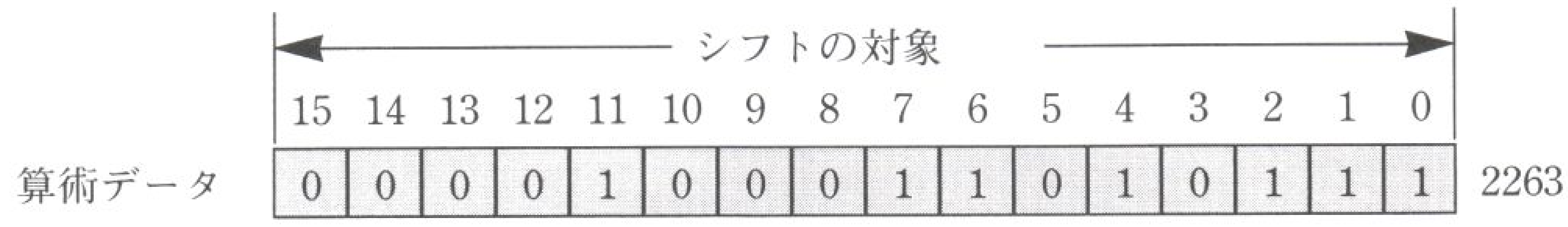
| | |
|---|---|
| `SLL    GR1,3` | 実効アドレスは3ですから、GR1の値を3ビット論理左シフトします。 |
| `SLL    GR0,0,GR1` | 実効アドレスはGR1の値ですから、GR0の値をGR1の値だけ論理左シフトします。 |

`SLL GR2,4,GR3` 実効アドレスは「4+GR3の値」ですから、GR2の値を「4+GR3の値」だけ論理左シフトします。

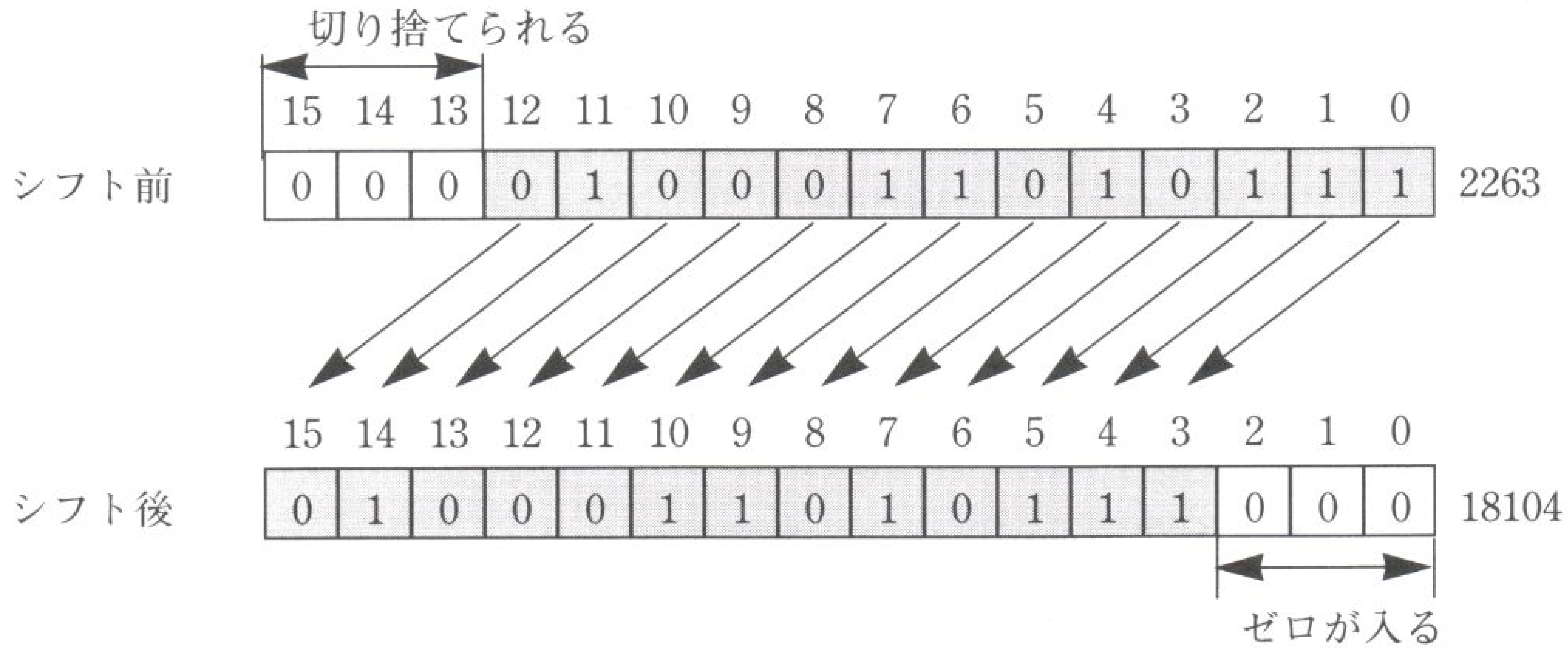
この他に構文としては、ラベルを使う方法がありますが、実際のプログラムの中ではほとんど使うことはないでしょう。

`SLL GR2,LBL,GR1` 実効アドレスは「LBLのラベルのアドレス＋GR1の値」となり、この分だけ論理左シフトします。

さて、今、次のデータがレジスタGR1にあるとします。



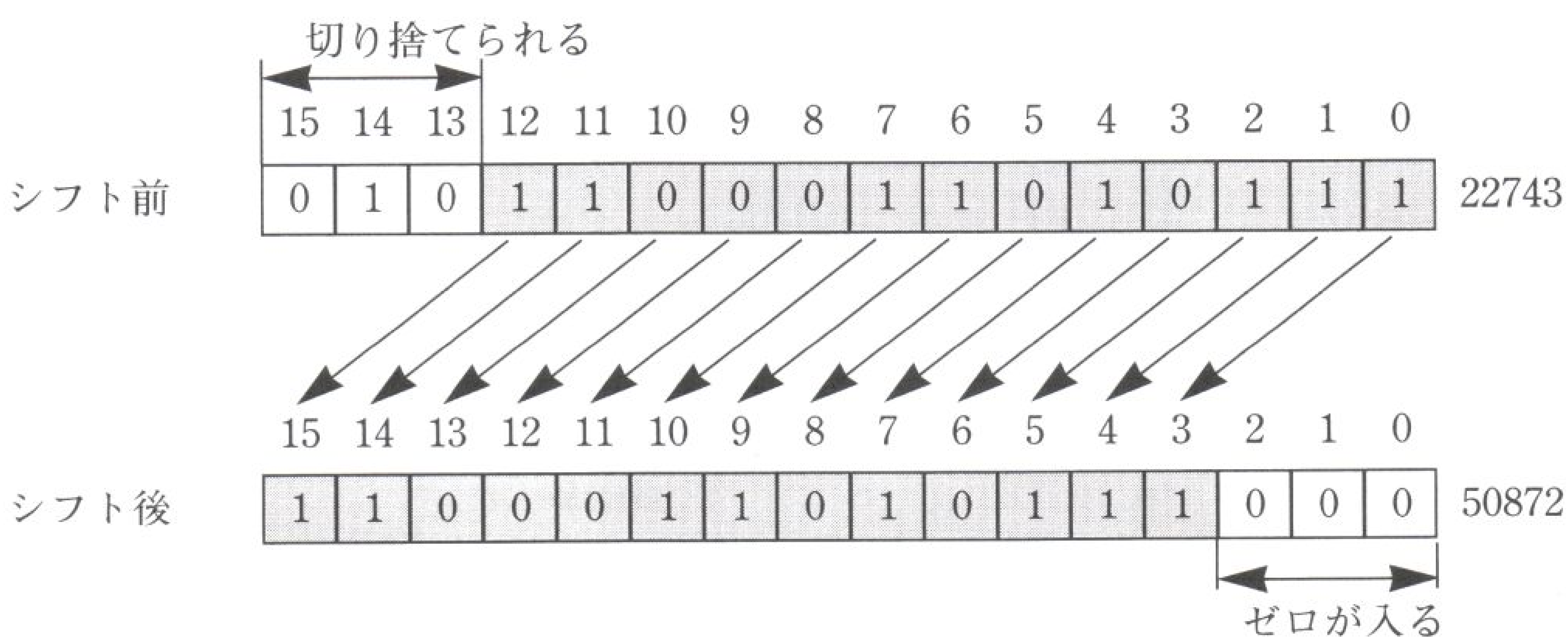

これを「SLL GR1,3」で3ビット左に論理シフトします。すると次のようになります。




ちょうど、18104＝2263×8ですから、元の数の$2^3$＝8倍になっています。
それでは、今度は、22743を3ビット左に論理シフトしてみましょう。



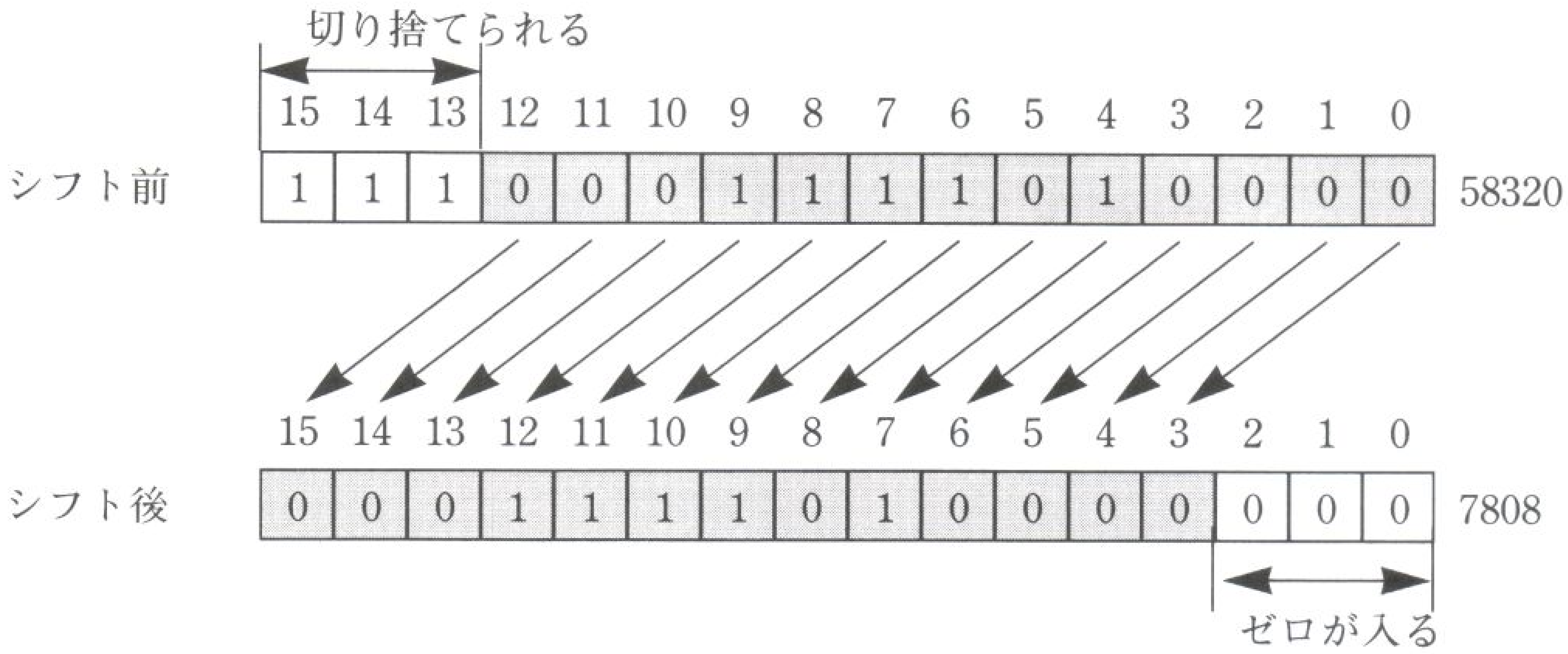

すると、次のようになります。



この場合には、上位3ビットがゼロではないので、その部分が切り捨てられるため、22743×8＝181944にはなりません。

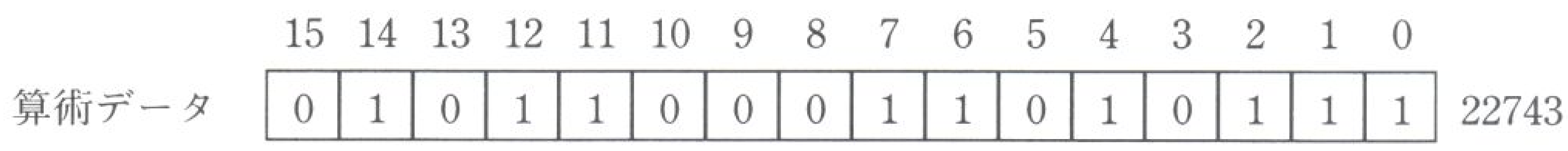
それでは、最上位ビットに1がある場合の論理左シフトはどのように処理されるのでしょうか。論理シフトの場合には、符号ビットを気にしないので、基本的にはまったく同じです。

それでは、今度は、58320を3ビット左に論理シフトしてみましょう。

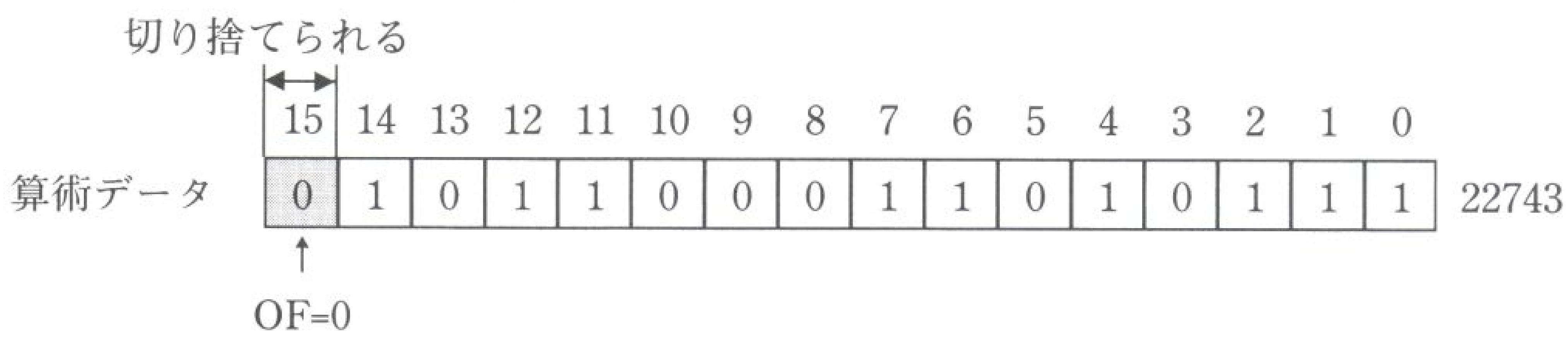



この場合、58320×8＝466560となるはずですが、上位3ビットの1が切り捨てられたため、7808となり8倍にはなっていません。

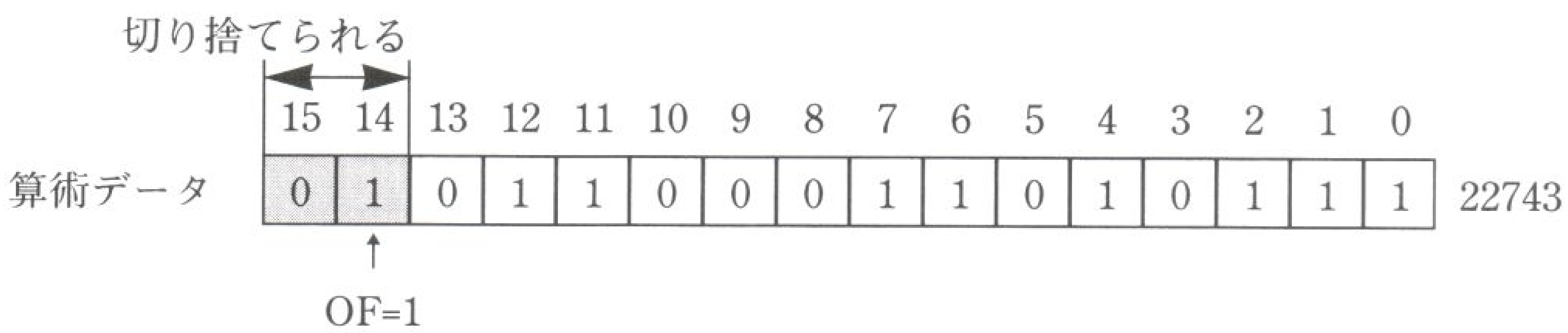
論理左シフト演算時のOF（オーバーフローフラグ）は、ちょっと分かりにくいので、ここで詳しく説明します。まず、22743という数値を1ビットおよび2ビット、62416という数値を4ビット論理左シフトしたときのオーバーフローフラグの設定について考えましょう。



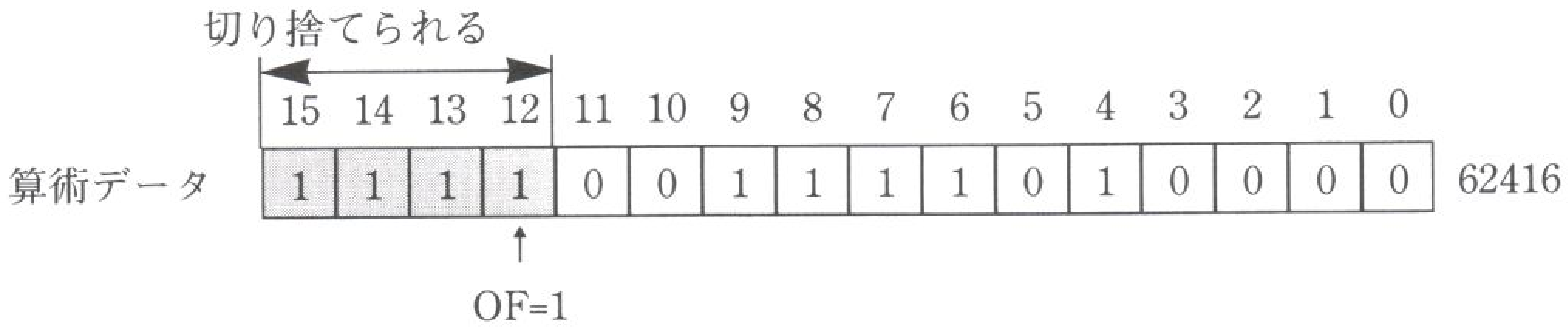

「仕様」では、「一番最後に押し出されたビットがOFにセットされる」と記述されています。22743を左に1ビット論理シフトすると、押し出されるのは15ビット目です。これがOFに入りますのでこの場合、OF＝0となります。




22743を左に2ビット論理シフトしたときに最後に押し出されるのは、14ビット目ですから、この場合、OF＝1となります。

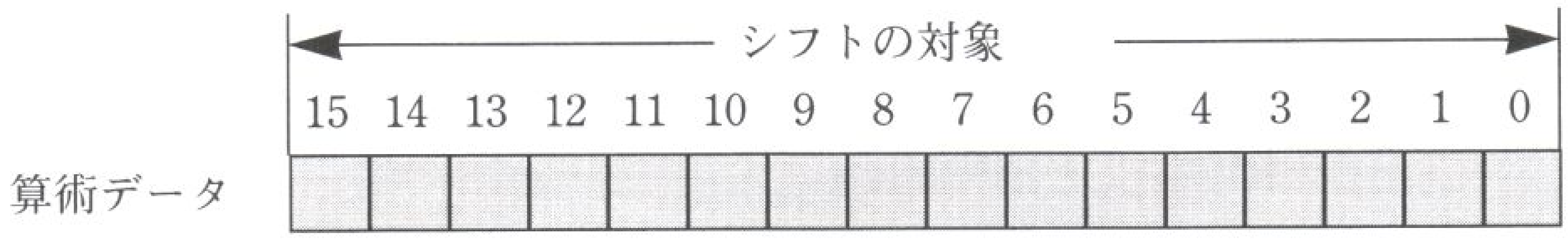


この規則は最上位ビットに1が立っていても同じです。62416を左に4ビット論理シフトしたときに最後に押し出されるビットは12ビット目ですから、その値がOFにセットされます。

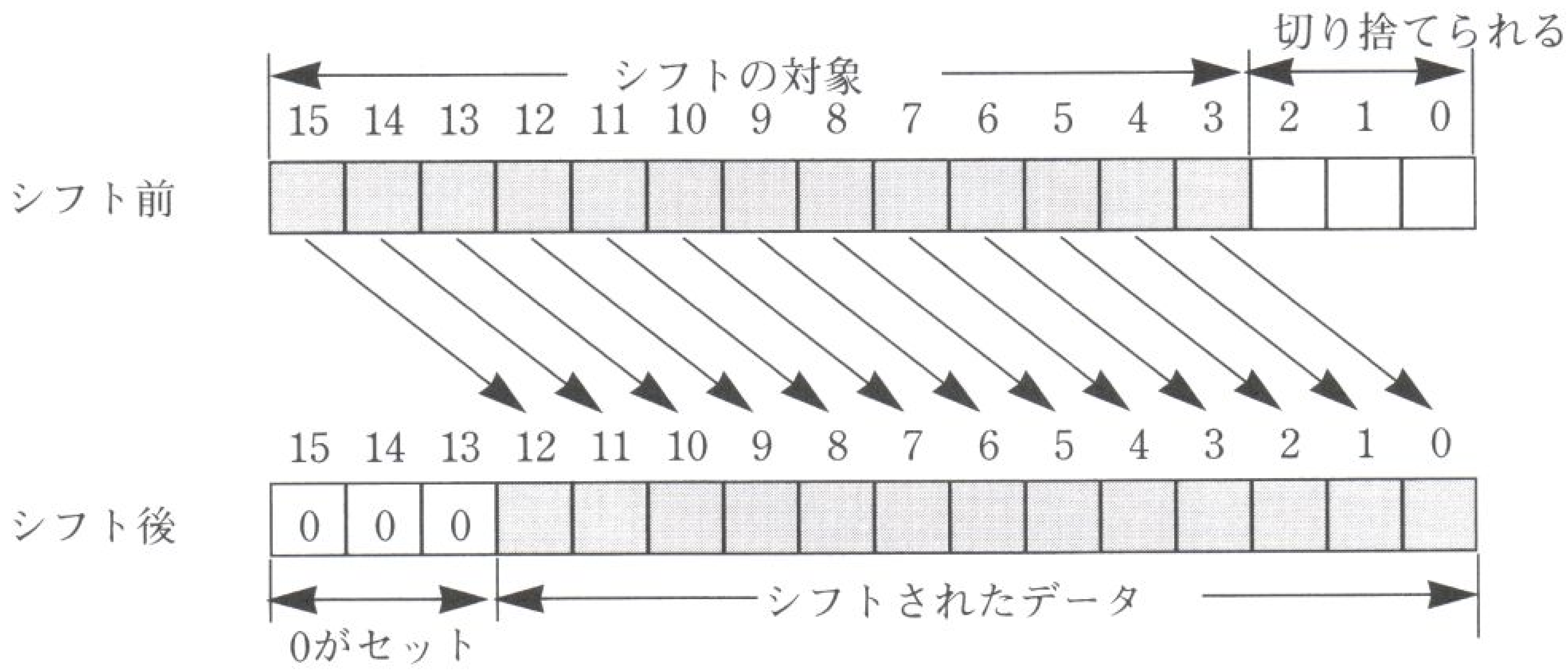



### 6.5.4 SRL命令

SRL命令は、実効アドレスで指定された数だけ、汎用レジスタの内容(値)を右に論理シフトします。論理シフトでは、すべてのビットが平等にシフトの対象になります。




また、シフトされて空いた上位ビットには常に0のビットが入ります。3ビット右に論理シフトすることを考えてみましょう。




それでは、まず、SRL命令の構文について見ていくことにしましょう。

| | |
|---|---|
| `SRL GR1,3` | 実効アドレスは3ですから、GR1の値を3ビット論理右シフトします。 |
| `SRL GR0,0,GR1` | 実効アドレスはGR1の値ですから、GR0の値をGR1の値だけ論理右シフトします。 |
| `SRL GR2,4,GR3` | 実効アドレスは「4+GR3の値」ですから、GR2の値を「4+GR3の値」だけ論理右シフトします。 |

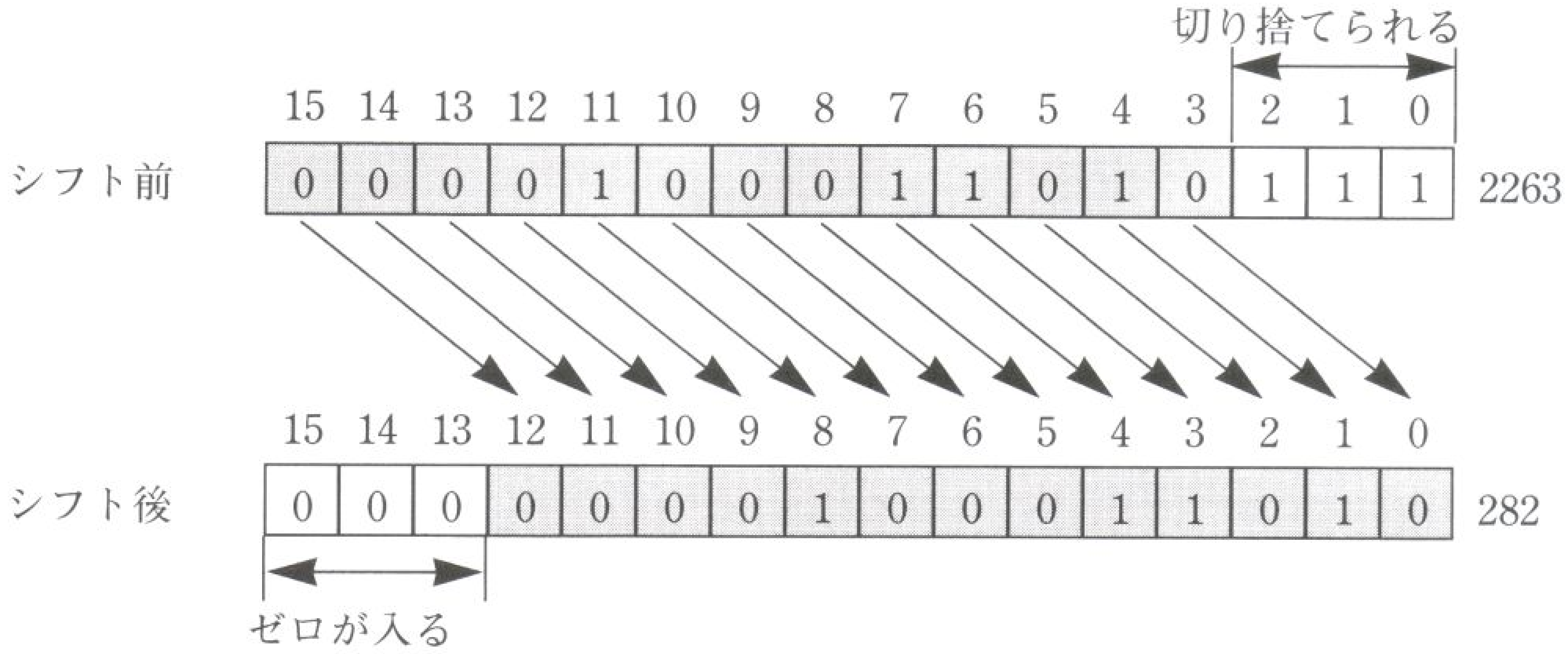
この他に構文としては、次の例のようにラベルを使う方法がありますが、実際のプログラムの中ではほとんど使うことはないでしょう。

| | |
|---|---|
| `SRL GR2,LBL,GR1` | 実効アドレスは「LBLのラベルのアドレス＋GR1の値」となり、この分だけ論理右シフトします。 |

さて、今、次のデータがレジスタGR1にあるとします。

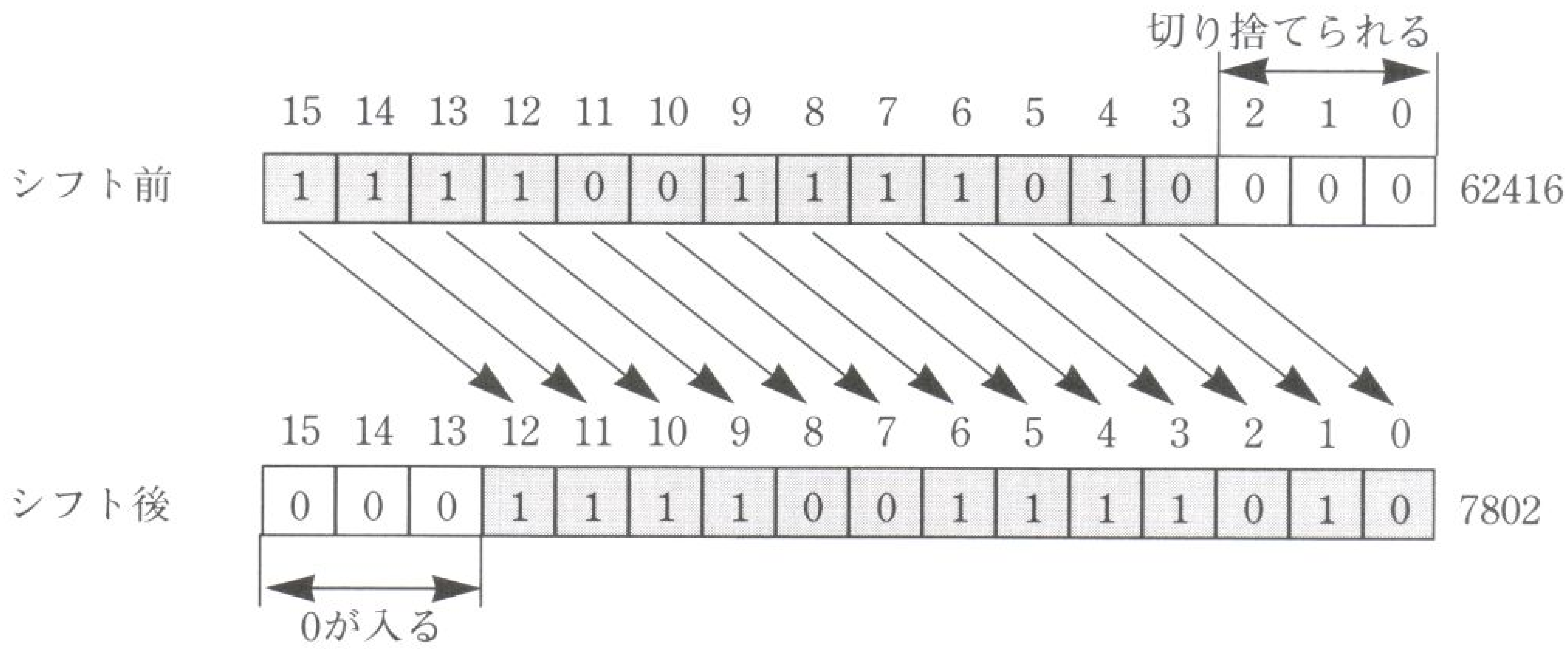



これを「SRL GR1,3」で3ビット右に論理シフトします。すると次のようになります。



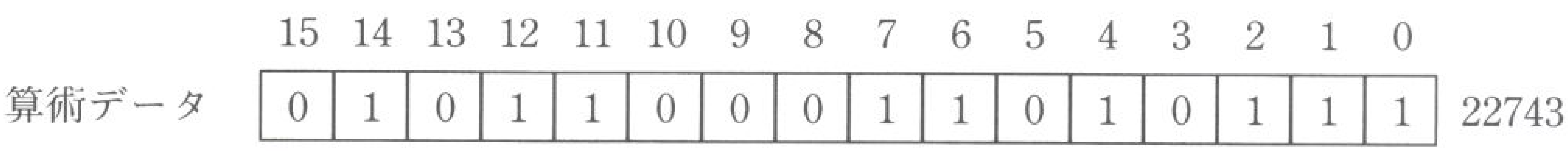

ちょうど、2263/8＝282.875ですから、整数の部分は元の数を$2^3$＝8で割った値になっています。小数点以下(0.875)が残っているのは切り捨てられた部分がゼロでないからです。切り捨てられた部分$111_{(2)}$＝7は、2263を8で割った余りになっています。

最上位ビットに1が立っていても、シフトで空いた上位ビットには常に0が入ります。それでは、今度は、次の数を3ビット右に論理シフトしてみましょう。




さて、62416/8＝7802となり、下位3ビットが切り捨てられても、ちゃんと8で割った値になっていることが分かると思います。

論理右シフト演算時のOF（オーバーフローフラグ）は、ちょっと分かりにくいので、ここで詳しく説明します。まず、22743という数値を1ビット、4ビット論理右シフトしたときのオーバーフローフラグの設定について考えましょう。




「仕様」では、一番最後に押し出されたビットがOFにセットされると記述されています。22743を

右に1ビット論理シフトすると、押し出されるのは0ビット目です。これがOFに入りますのでこの場合、OF＝1となります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 算術データ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 22743 |
| | | | | | | | | | | | | | | | | ↑<br>OF=1 | |

22743を左に4ビット論理シフトしたときに最後に押し出されるのは、3ビット目ですから、この場合、OF＝0となります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 算術データ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 22743 |
| | | | | | | | | | | | | | ↑<br>OF=0 | | | | |

# 6.6 分岐命令

## 6.6.1 分岐命令の概要

通常、命令は次の左のように上から下へ1つずつ実行されていきます。



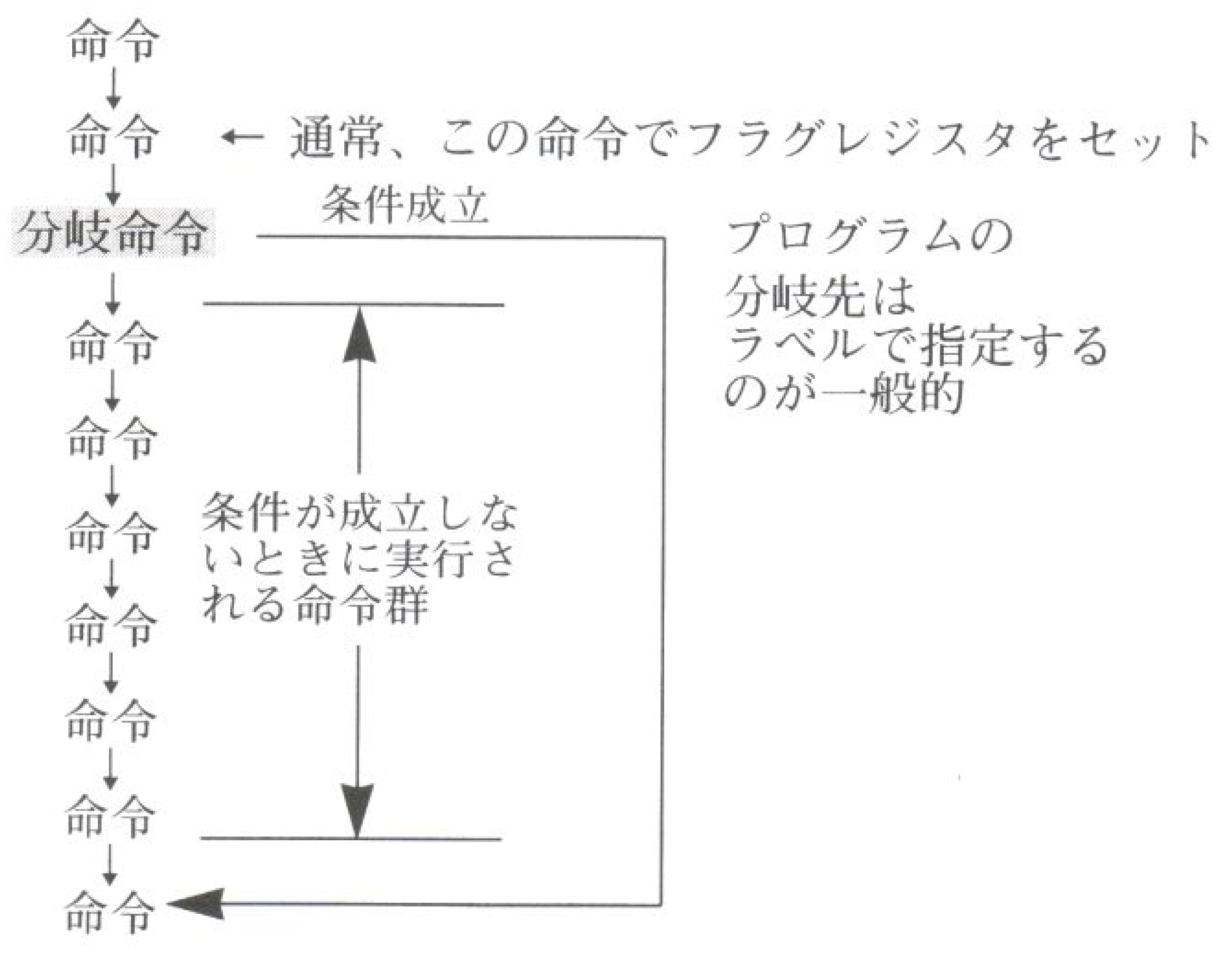

**分岐命令**は、フラグレジスタの値を見て、上の右のように実行の流れを変えるための命令で、次のものがあります。なお、実行の流れを変えることを**分岐する**といいます(5.4項参照)。

**表6-16：分岐命令**

| 命　　令 | 書き方 | | 命令の説明 |
|---|---|---|---|
| | 命令コード | オペランド | |
| 正分岐<br>Jump on PLus | JPL | adr[,x] | SF=0かつZF=0のとき、実効アドレスで指定されたアドレスに分岐する。 |
| 負分岐<br>Jump on MInus | JMI | adr[,x] | SF=1のとき、実効アドレスで指定されたアドレスに分岐する。 |
| 非零分岐<br>Jump on Non Zero | JNZ | adr[,x] | ZF=0のとき、実効アドレスで指定されたアドレスに分岐する。 |
| 零分岐<br>Jump on ZEro | JZE | adr[,x] | ZF=1のとき、実効アドレスで指定されたアドレスに分岐する。 |
| オーバーフロー分岐<br>Jump on OVerflow | JOV | adr[,x] | OF=1のとき、実効アドレスで指定されたアドレスに分岐する。 |
| 無条件分岐<br>unconditional JUMP | JUMP | adr[,x] | 無条件で、実効アドレスで指定されたアドレスに分岐する。 |

分岐命令は、条件判断を行ったり、ループを実現するのに用います。分岐しないときは、そのまま分岐命令の次の命令を実行します。なお、これらの命令はフラグレジスタには、影響を与えません。つまり、フラグレジスタには、元の値がそのまま残っています。

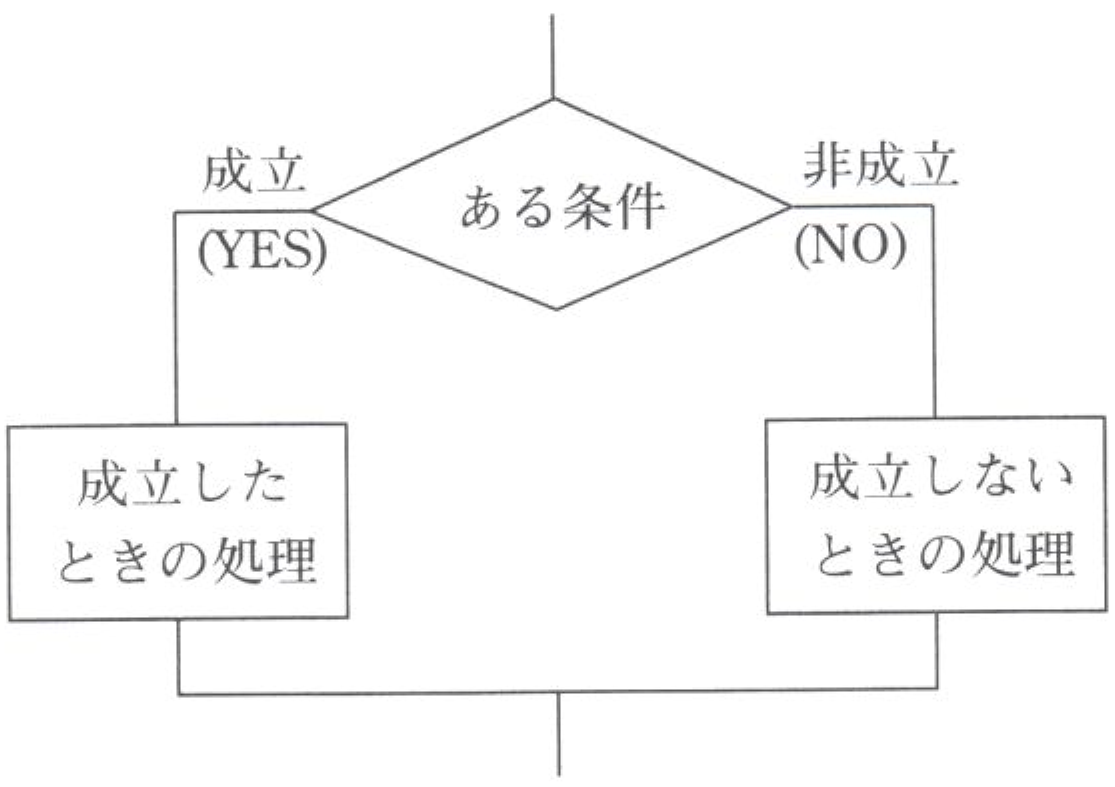
以下の例は、ある条件が成立したときに、前方の命令にジャンプするものです。すると、その分岐命令の後にある命令は、その条件の成立しないときだけ実行されます。



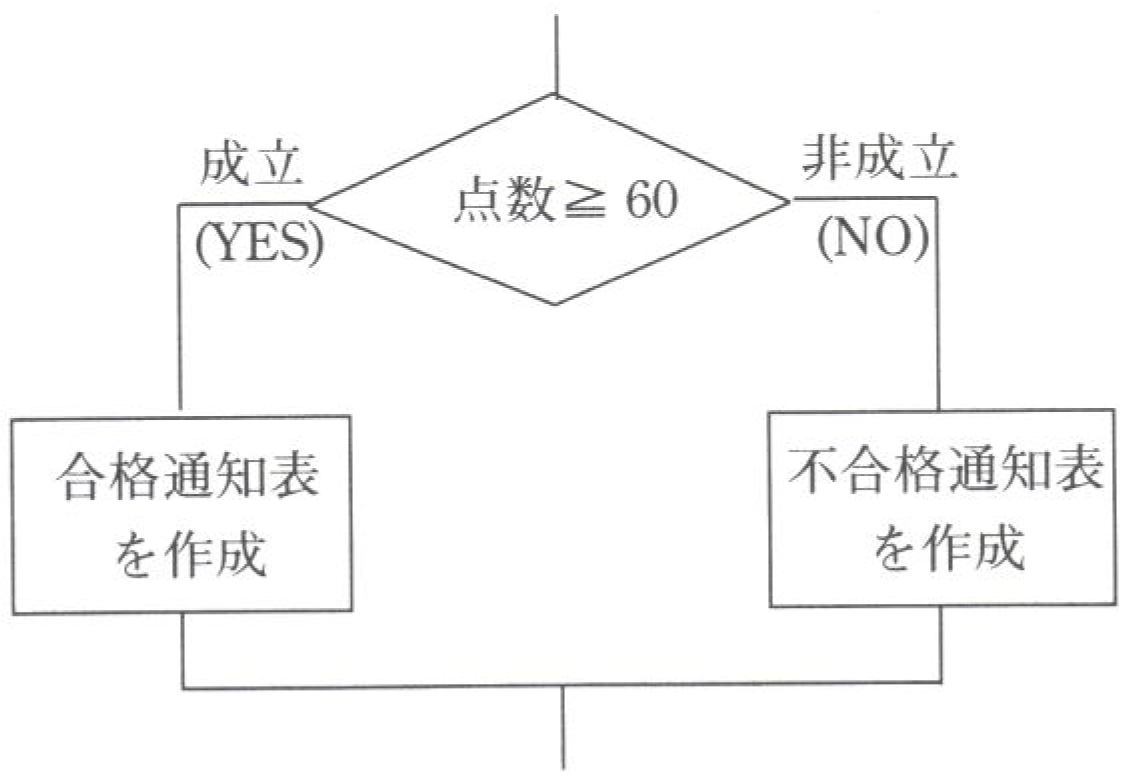
以下の例は、ある条件が成立したときに、ある命令群を実行し、そうでない場合、別の命令群を実行する例です。

フローチャートで示すと次のようになります。

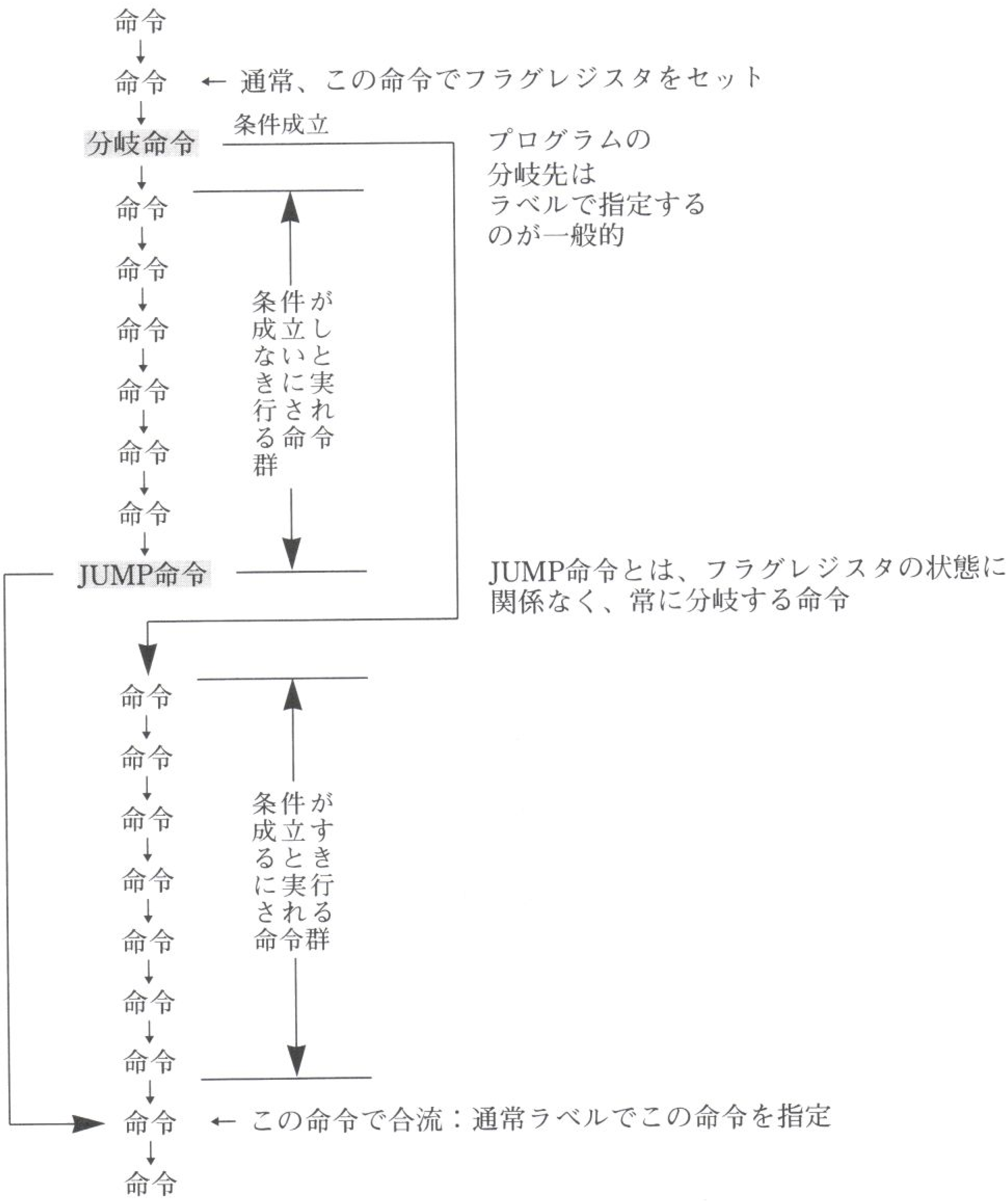
一般的な例




具体的な例

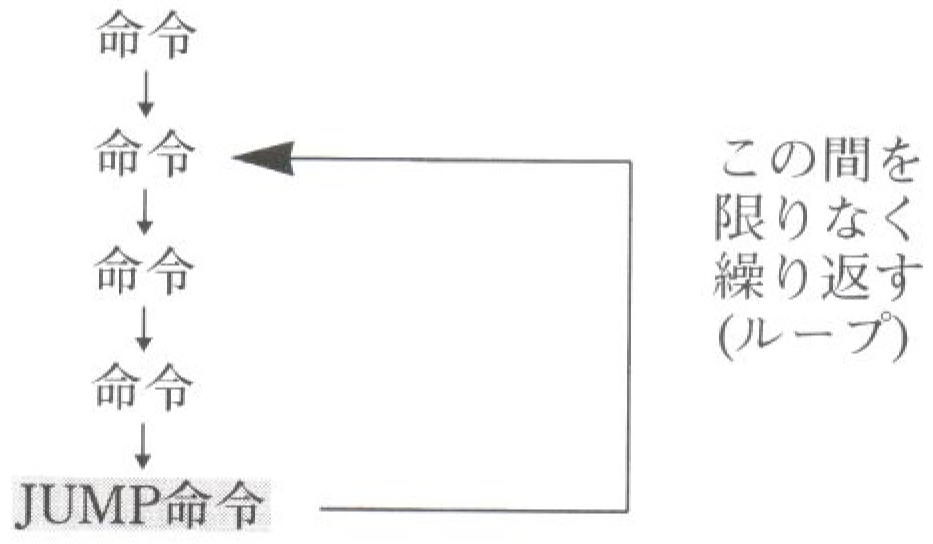



この流れは、次の命令で処理します。

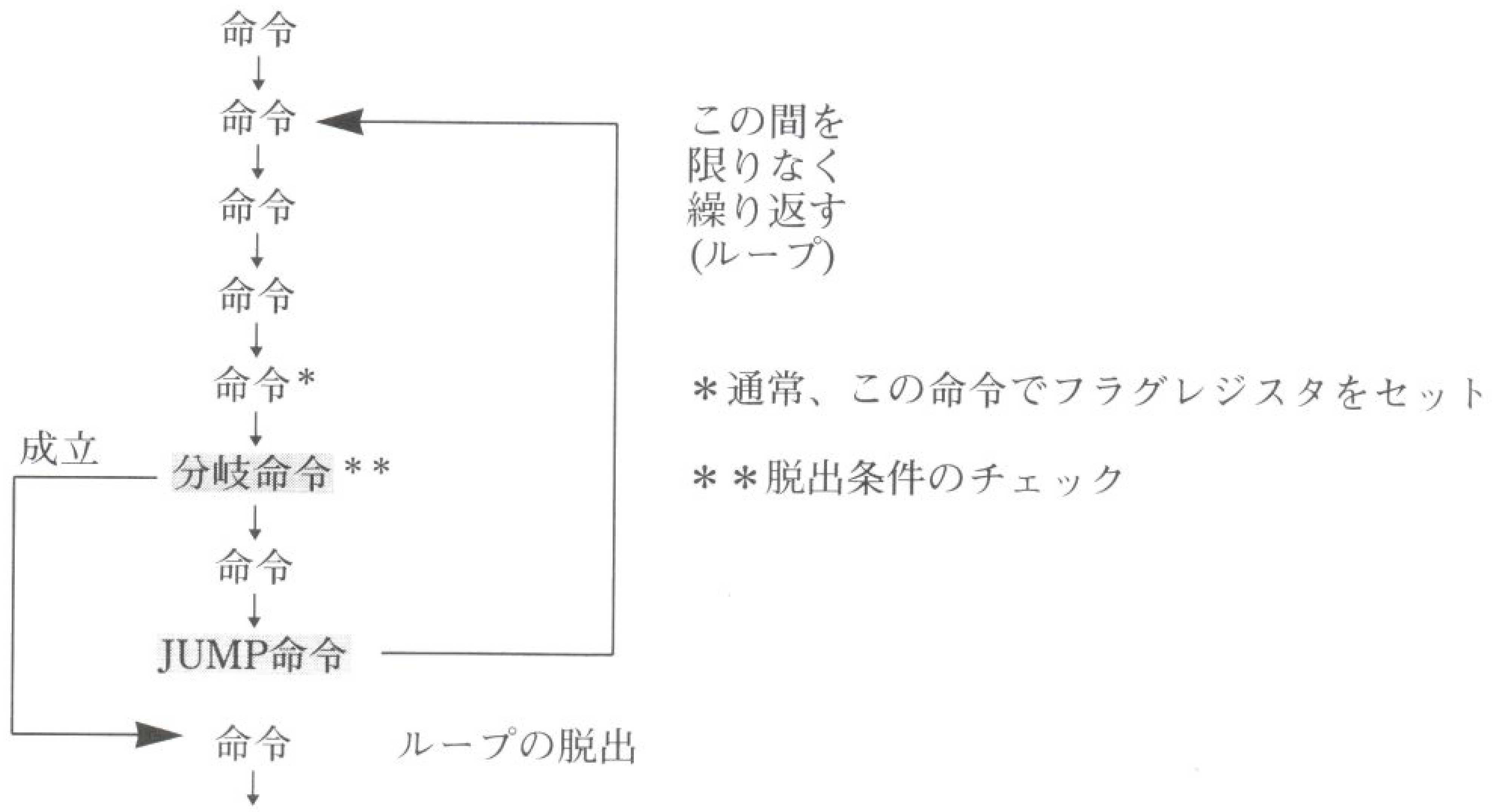


前頁の例では下方に分岐する流れを紹介しましたが、次に、分岐命令よりも上のどれかの命令へ戻ることによって、同じ命令群を何度も繰り返して実行する流れを紹介します。なお、同じ処理を何度も繰り返すことを**ループ**と呼びます。

次の命令群は、JUMP命令で、上の(すなわち、アドレスの小さい)命令に戻り、ループ(繰り返し処理)を実現しているものです。




しかし、この方法では、無限にループしてしまいます。これは**無限ループ**といって絶対に避けなければならないことです。そこで、そのループからの脱出を含めなければなりません。




上記以外にも、多くのループのパターンがありますが、それは、第9章で説明することにします。

さて、それでは、具体的な各分岐命令の動作について調べてみましょう。他の命令と異なり、分岐命令は、その命令の特徴だけを理解すれば、個々の命令を逐一学習しなくても、プログラムが組めるようになります。

分岐命令は大きく分けて次の**条件分岐命令**と**無条件分岐命令**の2つのパターンに分けられます。

**表6-17 分岐命令の種類**

| 種　類 | 説　明 | 対象となる命令 |
|---|---|---|
| 条件分岐命令 | フラグレジスタの値を見て分岐する(しないこともある) | JPL, JMI, JNZ, JZE, JOV |
| 無条件分岐命令 | 無条件で分岐する | JUMP |

条件分岐命令はフラグレジスタの値が次の時に分岐します。

**表6-18　各分岐命令とそれが分岐する時のフラグレジスタの値**

| 名　　前 | 命令 | OF | SF | ZF |
|---|---|---|---|---|
| 正分岐 | JPL | — | 0 | 0 |
| 負分岐 | JMI | — | 1 | — |
| 非零分岐 | JNZ | — | — | 0 |
| 零分岐 | JZE | — | — | 1 |
| オーバーフロー分岐 | JOV | 1 | — | — |

表6-18において、ーは、そのビットを無視することを意味します。たとえば、JPL命令はOFの値にかかわらずSF＝0かつZF＝0のときに分岐します。JMI命令はOF、ZFの値にかかわらず、SF＝1のときに分岐します。

さて、分岐命令は、どれも次の構文をしています。

```
分岐命令　adr[,x]
```

つまり、分岐命令のあとには、実効アドレスだけを記述します。JPL, JMI, JNZ, JZE, JOVの場合には、条件が成立したときだけ、その実効アドレスにジャンプします。JUMP命令は、常に指定された実効アドレスにジャンプします。

いくつか例をみてみましょう。

### (1) 絶対アドレスによる指定

絶対アドレスを指定して、主記憶装置の絶対番地に格納された命令にジャンプします。

```
分岐命令　100
```

次の3つの命令は、実効アドレスとして絶対アドレスを指定した例です。条件が成立すれば100番地にジャンプします。

```
JPL   100     SF=0かつZF=0なら100番地へジャンプ
JMI   100     SF=1なら100番地へジャンプ
JUMP  100     無条件で100番地へジャンプ
```

一般に、ジャンプ先のアドレスを主記憶装置の特定の番地に固定してしまうと、ローダで任意のアドレスにプログラムを配置できなくなるなど都合の悪いことが多いので、一般ユーザの作成するプログラムでは、お勧めはできません。

### (2) ラベルによる指定

この方法がもっとも一般的です。条件が成立すれば、ラベルのある命令にジャンプします。

```
            分岐命令    ラベル

ラベル      命令
```

次の例は、ZF＝1ならZEROPROCラベルへジャンプするプログラムです。

```
              JZE   ZEROPROC

ZEROPROC      命令
```

次の例は、ループを形成している例です。

```
               命令
LOOP           命令
               ∫                    通常、この命令でフラグレジスタをセット
               命令
               JMI    EXITLOOP      脱出条件(SF=1のときEXITLOOPへ)
               ∫
               JUMP   LOOP          LOOPへジャンプ(ループを形成)
EXITLOOP       命令
```

### (3) 指標レジスタに格納された番地へジャンプ

指標レジスタにラベルJUMPLBLのアドレスが格納されているとします。このとき、次の命令で条件が成立したとき(JUMP命令の場合は常に)、その指標レジスタに格納されたアドレスへジャンプします。

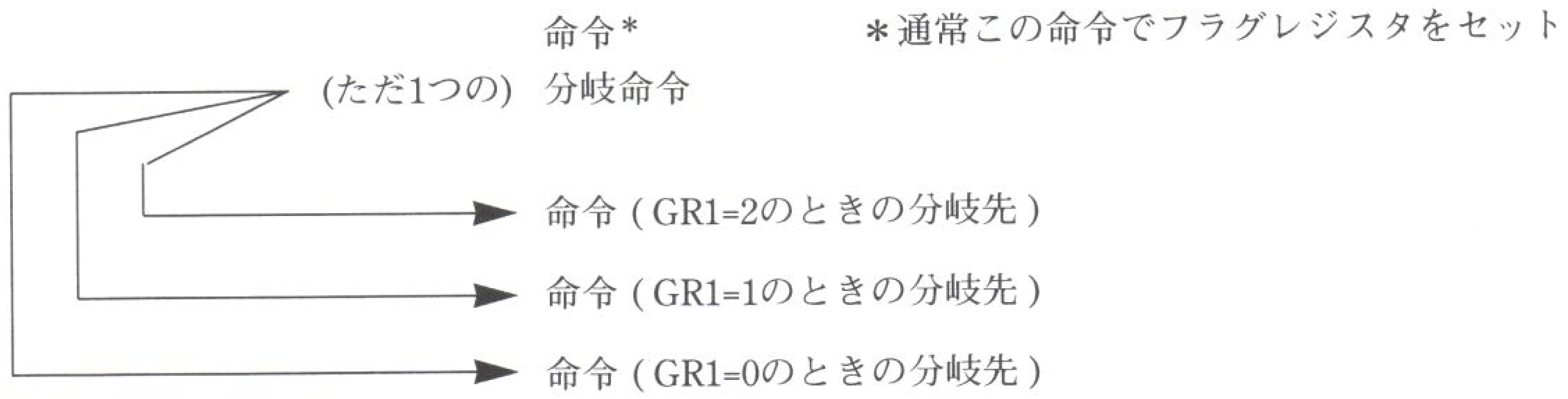
```
分岐命令  0,指標レジスタ
```

これは、「0,指標レジスタ」によって生成される実効アドレスが、まさに指標レジスタ内のアドレスであるからです。なお、ラベルのアドレスを指標レジスタにロードするには、LAD命令を使えば、よかったのでしたね。例を示しましょう。

```
LAD       GR7,JUMPLBL      GR7にJUMPLBLのアドレスをロード
            ∫                    (この間でGR7の値を変更していないとする)
JPL        0,GR7           SF=0かつZF=0ならば、GR7のアドレスにジャンプ
            ∫
JUMPLBL    命令
```

### (4) ジャンプテーブルの利用

今までは、ジャンプ先はただ一つでした。しかし、プログラミングによっては、あるレジスタの値によって、複数のジャンプ先に分岐したい場合もあります。次の例は、GR1の値によってジャンプ先が変わっています。




もちろん、条件判断を複数組み合わせて、この処理を実現することもできますが、アセンブラ言語は効率が第一です。この命令をただ一つの分岐命令で実現すればとてもコンパクトなプログラムになります。

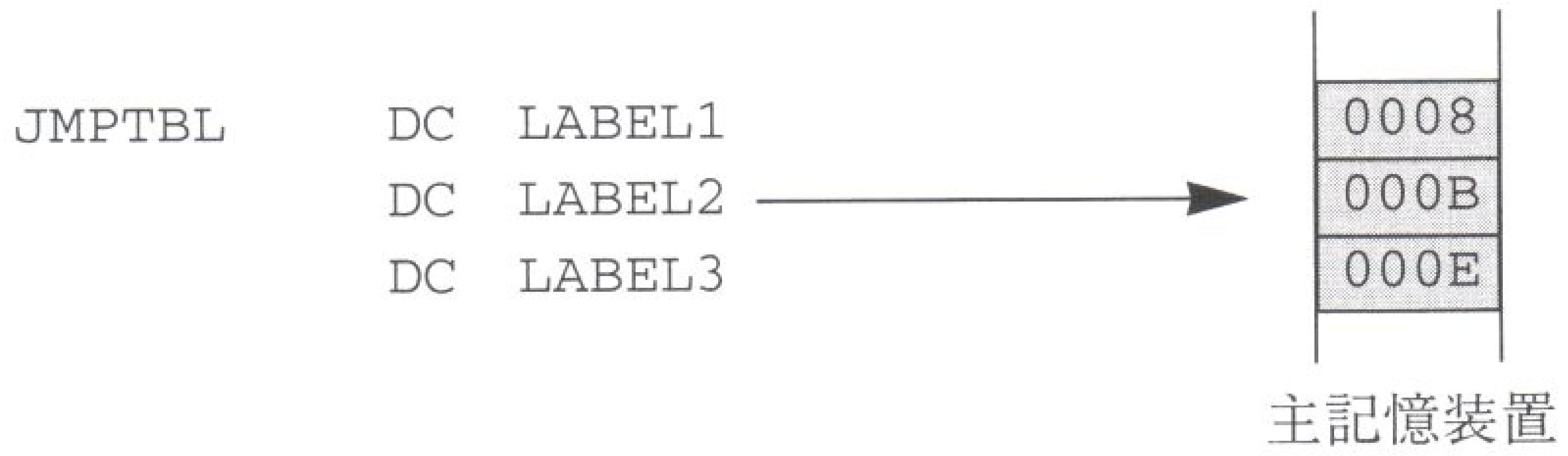
これを実現するには、2つの方法があります。なお、以下の方法では、例としてJUMP命令を使っていますが、これ以外の命令(たとえば、JPLやJMI)であっても方法は同じです。

### (a) アドレスを表として主記憶領域に格納し、それを利用する方法

このプログラムは、ちょっと複雑です。まず、このプログラム中に次の3つのラベル(LABEL1, LABEL2, LABEL3)が定義されているとしましょう。

```
LABEL1      命令
LABEL2      命令
LABEL3      命令
```

さて、LABEL1が8番地、LABEL2がB番地、LABEL3がE番地、であったとしましょう。このとき、この3つのラベルのアドレスを、この順で保持するようにするには、次の命令を使います。




または、次のように1桁で記述することもできます。

```
JMPTBL      DC   LABEL1,LABEL2,LABEL3
```

DC命令のオペランドにラベルを書くと、そこにそのラベルのアドレスが置かれます。さて、それでは、GR1の値が0ならば、JMPTBLの最初のアドレス(0008)を取り出し、そこにジャンプし、GR1の値が1ならば、JMPTBLの2番目のアドレス(000B)を取り出し、そこにジャンプし、GR1の値が2ならば、JMPTBLの3番目のアドレス(000E)を取り出し、そこにジャンプするプログラムを考えてみましょう。まず、次の命令で、GR7にジャンプするアドレス(0008あるいは、000B、000E)を取り出します。

```
LD  GR7,JMPTBL,GR1
```

これは、「JMPTBLラベルの領域のアドレス＋GR1の値」番地の内容をGR7にロードする命令でし

た。したがって、GR1が0なら0008がGR7へ、GR1が1なら000BがGR7へ、GR1が2なら000EがGR7へ、ロードされます。そして、次の命令でロードしたアドレスへジャンプします。

```
JUMP  0,GR7
```

このプログラムを組込んだ簡単な例を示します。GR1が0ならLABEL1へジャンプしてGR0に0をロード、GR1が1ならLABEL2へジャンプしてGR0に1をロード、GR1が2ならLABEL3へジャンプしてGR0に2をロードする単純なプログラムです。

アドレスを示すために、アセンブラリストといっしょに示します。なお、プログラムの中で機械語のアドレスを適当に空ける(たとえば、4～5番地のJUMP命令と8番地の命令の間に2語分空ける)ためにNOP命令を用いています。これは、6.9.2項で説明しますが、「何もしない」という命令です。NOP命令は、この例のように、必要はないんだけれども、何か命令を書いておきたいという場合に使います。

このNOP命令は、次の(b)に示すプログラムJMPTBL2でも使っており、そちらの方がNOP命令の使用例としては分かりやすいかも知れません。

なお、プログラムJMPTBL1.caslは、このプログラムが格納されているファイル名です。このファイルは本書に付属のCD-ROMのSAMPLEフォルダの中のchap6フォルダの中に入っています。これをハードディスクなどにコピーしてから本書に付属のCASLDVにかけてください。なお、本書に付属のCASLDVにかけると「5行目のNOP命令が実行されない」という警告メッセージが出力されますが、ここでは無視して実行してください。

```
    Machine        Source List (JMPTBL1.casl)
Adr.    Code   Line    Program
               1    JMPTBL1    START
   0   1210    2               LAD GR1,1    JMPTBLの何個めのアドレスにジャンプするか
   1   0001
   2   1071    3               LD   GR7,JMPTBL,GR1   GR7 ← JMPTBLのGR1番目
   3   0011
   4   6407    4               JUMP   0,GR7   GR7番地へジャンプ
   5   0000
   6   0000    5               NOP            ;まったく意味がないがここを少しあけたいために
   7   0000    6               NOP            ;NOP命令を入れた
   8   1200    7    LABEL1     LAD GR0,0    GR1=0のとき、ここに飛んでくるはず
   9   0000
   A   8100    8               RET
   B   1200    9    LABEL2     LAD GR0,1    GR1=1のとき、ここに飛んでくるはず
   C   0001
   D   8100   10               RET
   E   1200   11    LABEL3     LAD GR0,2    GR1=2のとき、ここに飛んでくるはず
   F   0002
  10   8100   12               RET
  11   0008   13    JMPTBL     DC   LABEL1    ジャンプ用のアドレステーブル
  12   000B   14               DC   LABEL2
  13   000E   15               DC   LABEL3
              16               END
```

この例では、GR1に1がロード(2行目)されているので、最終的にLABEL2へ分岐してGR0には1がロードされて終了するはずです。

このプログラムを0番地からロードして実行してみましょう。

```
PR   Instruction        SP  OSZ GR0  GR1  GR7  Explanation
0000 LAD  GR1,0001      7530 --- 8029 0001 F2AE [h:0001, s:1, u:1]
0002 LD   GR7,0011,GR1  7530 --- 8029 0001 000B 0012:[h:000B, s:11, u:11]
0004 JUMP 0000,GR7      7530 --- 8029 0001 000B jump to 000B
000B LAD  GR0,0001      7530 --- 0001 0001 000B [h:0001, s:1, u:1]
000D RET                7530 --- 0001 0001 000B Return to OS
```

実行結果をみると、ちゃんとB番地へジャンプしていることが分かります。

### (b) 無条件ジャンプのプログラムを並べ、そのどれかへジャンプする方法

この考え方の基本は次の通りです。まず、次のようにJUMP命令だけのプログラムを作成します。これが今、次のアドレスにあるとしましょう(*1)。

```
6 番地  JMPENTRY  JUMP   LABEL1
8 番地            JUMP   LABEL2
A 番地            JUMP   LABEL3
```

この状態で、もし、次の命令を実行したとするとどうなるでしょう。

```
JUMP   JMPENTRY
```

当然のことですが、JMPENTRYにジャンプし、その中の命令「JUMP LABEL1」を実行して、LABEL1にジャンプして、LABEL1にある機械語命令を実行します。それでは、GR1に2が格納されているときに、次の命令を実行したら、どのようになるでしょうか。

```
JUMP   JMPENTRY,GR1
```

この命令によって生成される実効アドレスは、「JMPENTRYのラベルのアドレス＋GR1の値」ですから「6番地＋2」＝「8番地」ということになります。したがって、8番地にジャンプし、そこの「JUMP LABEL2」が実行されます。このプログラムでは1つ目のラベルに分岐する場合、GR1には0、2つ目のラベルに分岐する場合GR1には2、3つ目のラベルに分岐する場合RG1には4を入れる必要があります。

なお、このプログラムでも、2〜3番地のJUMP命令と6〜7番地のJUMP命令の間を空けるためにNOP命令を使っています。もちろん、空けなくても良いのですが、そうしないと本当にジャンプしたのか次の命令を実行したのかトレース結果だけではわからなくなります。

さて、それでは、このプログラムを実行してみましょう。なお、このプログラム(JMPTBL2.casl)も前のプログラム(JMPTBL1.casl)と同じように本書の付属のCASLDVにかけると警告メッセージが出力されますが、無視して実行してください。

(＊1) JUMP命令の語長は2語です。

```
  Machine      Source List (JMPTBL2.casl)
 Adr.  Code        Line     Program
                 1  JMPTBL2   START
    0  1210      2            LAD   GR1,2   JMPENTRYの何個めのアドレスにジャンプするか
    1  0002
    2  6401      3            JUMP JMPENTRY,GR1    ジャンプテーブルへジャンプ
    3  0006
    4  0000      4            NOP
    5  0000      5            NOP
    6  6400      6  JMPENTRY  JUMP LABEL1
    7  000E
    8  6400      7            JUMP LABEL2
    9  0011
    A  6400      8            JUMP LABEL3
    B  0014
    C  0000      9            NOP
    D  0000     10            NOP
    E  1200     11  LABEL1    LAD   GR0,0  GR1=0のとき、ここに飛んでくるはず
    F  0000
   10  8100     12            RET
   11  1200     13  LABEL2    LAD   GR0,1  GR1=2のとき、ここに飛んでくるはず
   12  0001
   13  8100     14            RET
   14  1200     15  LABEL3    LAD   GR0,2  GR1=4のとき、ここに飛んでくるはず
   15  0002
   16  8100     16            RET
                17            END
```

このプログラムを0番地からロードして実行すると、次のようになりました。

```
PR   Instruction       SP OSZ GR0  GR1  GR2  GR3  GR4  GR5  GR6  GR7  Explanation
0000 LAD   GR1,0002   7530---8029 0002 98BE E784 CAE1 BD6C ACD6 F2AE [h:0002, s:2, u:2]
0002 JUMP  0006,GR1   7530---8029 0002 98BE E784 CAE1 BD6C ACD6 F2AE jump to 0008
0008 JUMP  0011       7530---8029 0002 98BE E784 CAE1 BD6C ACD6 F2AE jump to 0011
0011 LAD   GR0,0001   7530---0001 0002 98BE E784 CAE1 BD6C ACD6 F2AE [h:0001, s:1, u:1]
0013 RET              7530---0001 0002 98BE E784 CAE1 BD6C ACD6 F2AE Return to OS
```

### 6.6.2 JOV命令

JOV命令は、OF(オーバーフローフラグ)を使用する唯一の命令ですが、OFについて、これまできちんと説明してきませんでした。そこで、本項でOFについて詳細に説明することにします。

OFの値は、表6-19に示す各命令によってのみ変化します。シフト命令以外の命令の場合、演算結果がそのデータ表現で表現できる範囲を超えた場合にOFは1になり、それ以外の場合にはOFは0になります。たとえば、GR1に32767(符号付きデータで表現できる最大値)、GR2に3が入っているときに、「ADDA GR1,GR2」を実行すると、数学的には32770になりますが、この値は符号付きデータでは表現できないのでOF=1となります。ちなみに、GR1の値は符号付きデータ表現で−32766となります。一方、GR1,GR2に同じ値(GR1=32767,GR2=3)が入っているときに、「ADDL GR1,GR2」を実行すると

正しく32770となり、OFは0となります。しかし、GR1に40000(符号無し表現、16進数では9C40(16))、GR2に30000(符号無し表現、16進数では7530(16))をセットして「ADDL GR1,GR2」を実行すると結果は70000となるはずですが、この値は符号無しデータでも表現できないのでOF=1となります。ちなみに、この場合、GR1には1170(16)=4464が格納されます。これは、9C40(16)+7530(16)=11170(16)という計算結果の下位4バイトの値になっています。ところで、GR1,GR2に同じ値が入っているときに、「ADDA GR1,GR2」を実行するとどうなるでしょうか。同じ値といっても、GR1に格納されている9C40(16)は符号付き表現では－25536であり、GR2に格納されている7530(16)は30000です。したがって、GR1+GR2=－25536+30000=4464と正しく計算され、OF=0となります。

OFは、計算結果がアンダーフローした場合にも発生します。GR1=－32768, GR2=－10として「ADDA GR1,GR2」を実行すると、数学的には－32778になりますが、この値は符号付きデータでは表現できませんので、やはりOF=1となります。

このように、シフト命令以外の命令では、演算結果がオーバーフローあるいはアンダーフローのときにOF=1となり、それ以外はOF=0となります。

シフト命令の場合には、6.5節で説明するように、シフト演算の結果、最後に押し出されたビットが1のときにOF=1となり、最後に押し出されたビットが0のときにOF=0となります。

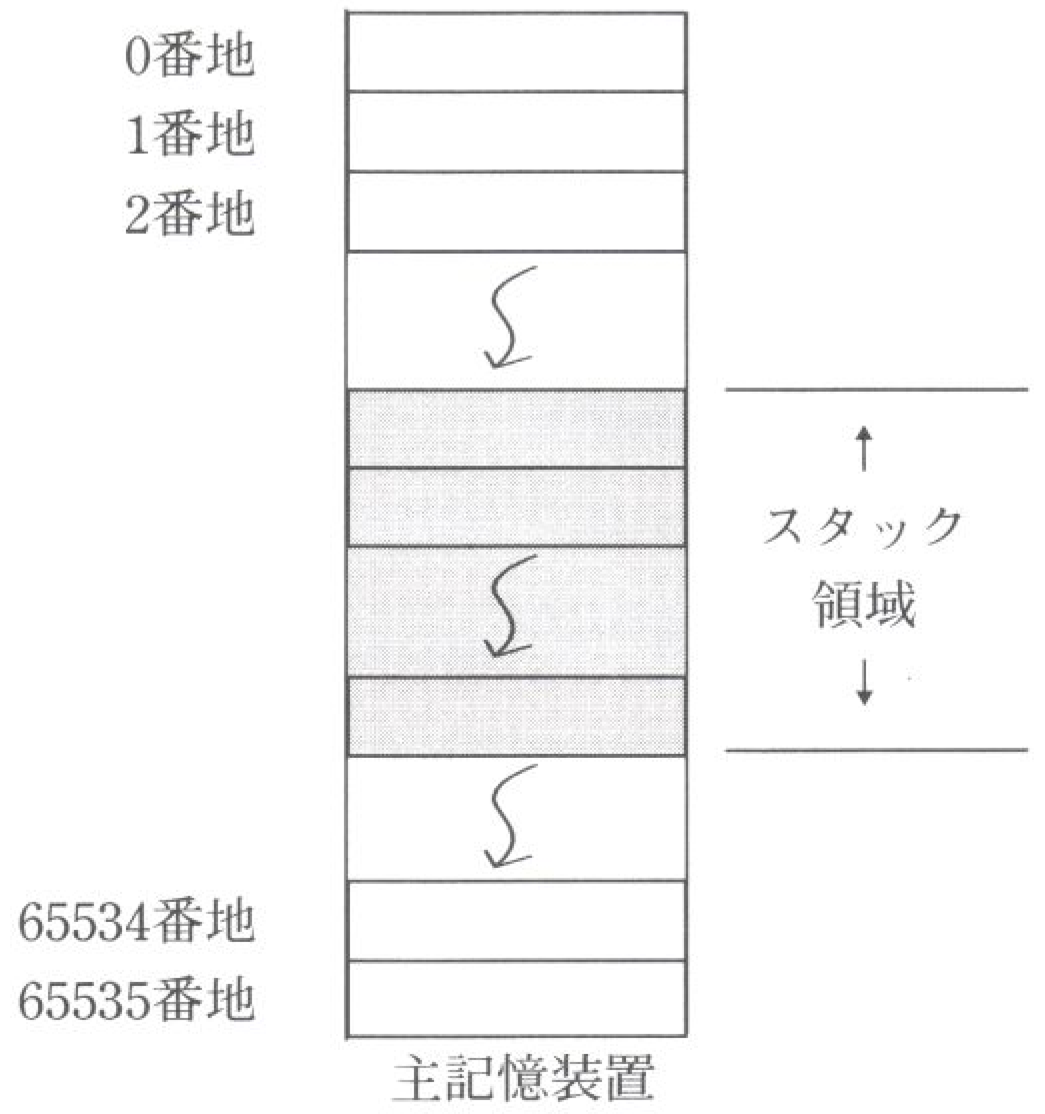
これらの関係を以下の表に示します。

**表6-19 オーバーフローフラグを設定する命令とそのときの条件**

| 命令語 | OF=1 | OF=0 |
|---|---|---|
| ADDA<br>SUBA | 演算結果が<br>－32768～32767<br>の範囲外のとき | 演算結果が<br>－32768～32767<br>の範囲内のとき |
| ADDL<br>SUBL | 演算結果が<br>0～65535<br>の範囲外のとき | 演算結果が<br>0～65535<br>の範囲内のとき |
| SLA, SRA | 最後に押し出された<br>ビットが1のとき<br>(符号ビットは対象外) | 最後に押し出された<br>ビットが0のとき<br>(符号ビットは対象外) |
| SLL, SRL | 最後に押し出された<br>ビットが1のとき<br>(全ビットが対象) | 最後に押し出された<br>ビットが0のとき<br>(全ビットが対象) |

# 6.7 スタック操作命令

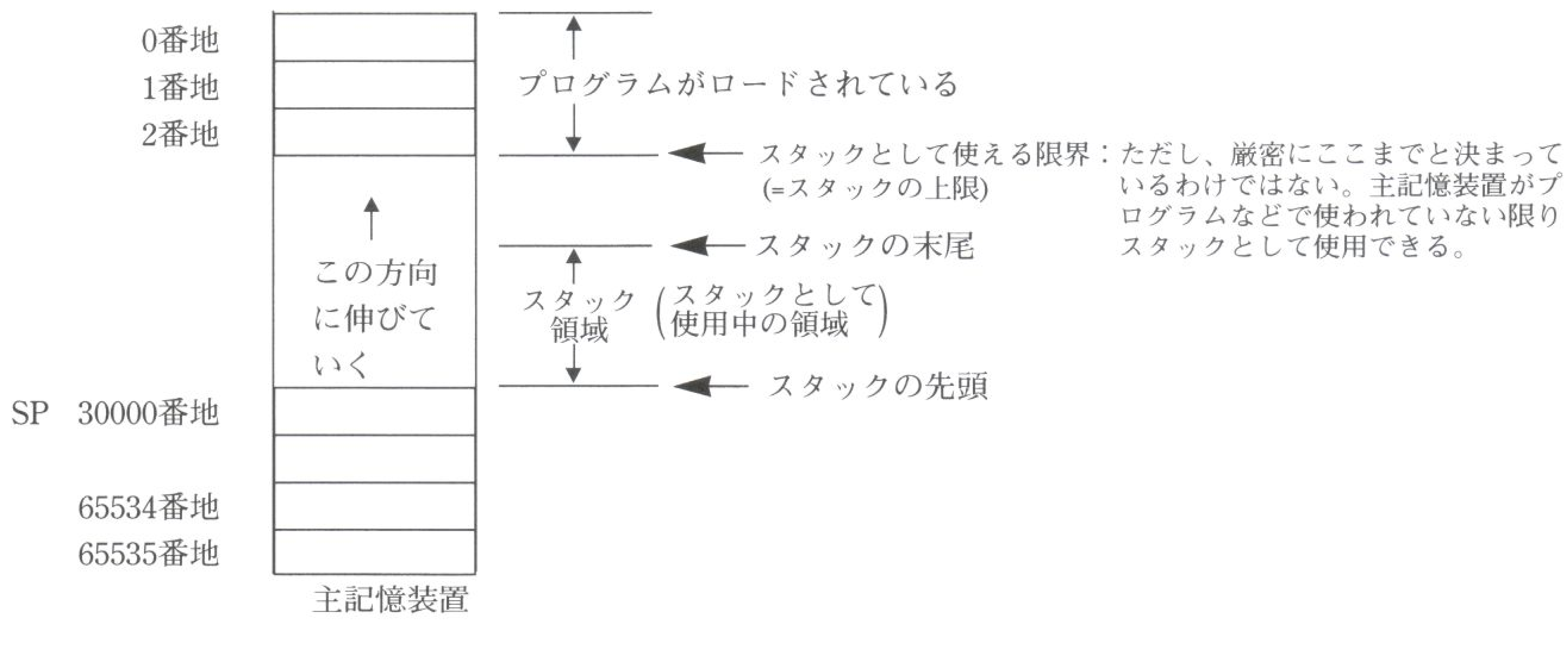
COMET IIには、主記憶装置があり、0語〜65535語までの領域があります。この主記憶装置の一部を**スタック**(stack)という領域として使います。これを**スタック領域**と呼びます。




もちろん、主記憶装置に印などが付いているわけではありません。プログラムの一つの約束事として、主記憶装置の特定の領域をスタック領域として使おうと決めておくのです。残念ながら、COMET IIでは、機械語を使って、スタック領域を決定することができません(一般のコンピュータでは、機械語を使ってスタック領域を決めることができます)。

しかし、どの領域がスタックであるかを本書に付属のCASLDVでは簡単に知ることができます。それは、SP(スタックポインタ)の値を見ればよいのです。

本書に付属のCASLDVでは、スタック領域の開始位置を7530(16)＝30000番地からと決めています(*1)。ただし、スタックは、アドレスの減る方向に使っていきますので、たとえば、プログラムが0番地からロードされている場合には、次のような配置関係になります(*2)。




(*1) もちろん、この値は変更することができます。メニューから「オプション」→「実行時のオプションの指定」を選び、「スタックの先頭アドレス」の値を変更してください。

(*2) これをメモリマップと呼びますが、CASLDVでは、メニューから「表示」→「メモリマップとロード状況」を選ぶと表示されます。

さて、本書のCASLDVのトレースを見ると、SPの最初の値が7530(16)になっていることに気付くでしょう。

```
PR   Instruction       SP   OSZ  GR0  GR   GR2  GR3  GR4  GR5  GR6  GR7   Explanation
0000 LAD   GR1,0002   7530 --- 8029 0002 98BE E784 CAE1 BD6C ACD6 F2AE  [h:0002, s:2, u:2]
                       ↑
```

SP(スタックポインタ)の値

さて、それでは、スタックとは、どのようなものなのでしょうか。それは、実にうまく主記憶装置を使う巧妙なやり方を提供します(*1)。

スタック操作には、次に示す2つの命令が用意されています。

**表6-20 スタック操作命令**

| 命令 | 書き方 | | 命令の説明 |
|---|---|---|---|
| | 命令コード | オペランド | |
| プッシュ PUSH | PUSH | adr[,x] | SP ← (SP) −L 1<br>(SP) ← 実効アドレス |
| ポップ POP | POP | r | r ← ( (SP) )<br>SP ← (SP) +L 1 |

−Lは論理減算、+Lは論理加算を意味します。

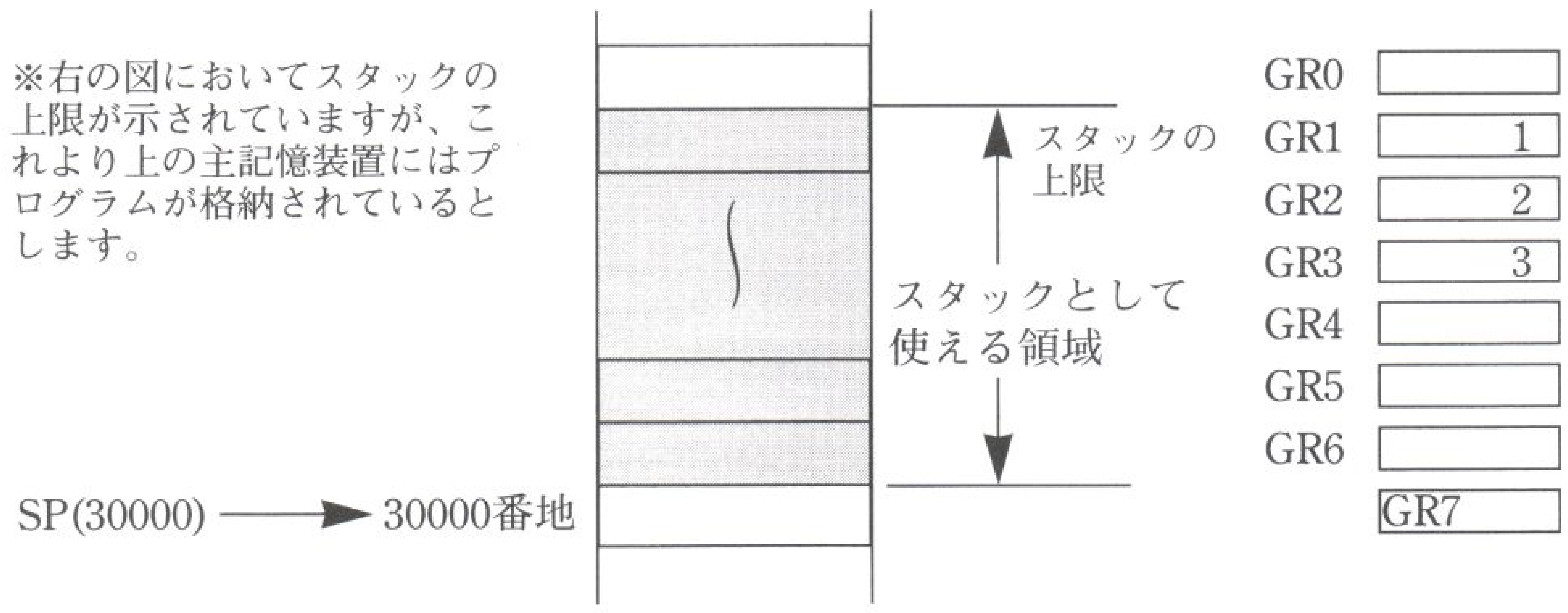
PUSH命令は、スタック上にデータを積みます(これを**プッシュダウン**といいます)。POP命令は、スタック上からデータを取り出して、レジスタにロードします(これを**ポップアップ**といいます)。

次の3つの命令を見てください。

```
PUSH    0,GR1
PUSH    0,GR2
PUSH    0,GR3
```

この最初の「PUSH 0,GR1」命令は、実効アドレスが「0+GR1」ですからレジスタGR1の内容をスタックにプッシュします。この命令の処理前は、SPと主記憶装置の関係が次のようになっているとしましょう。

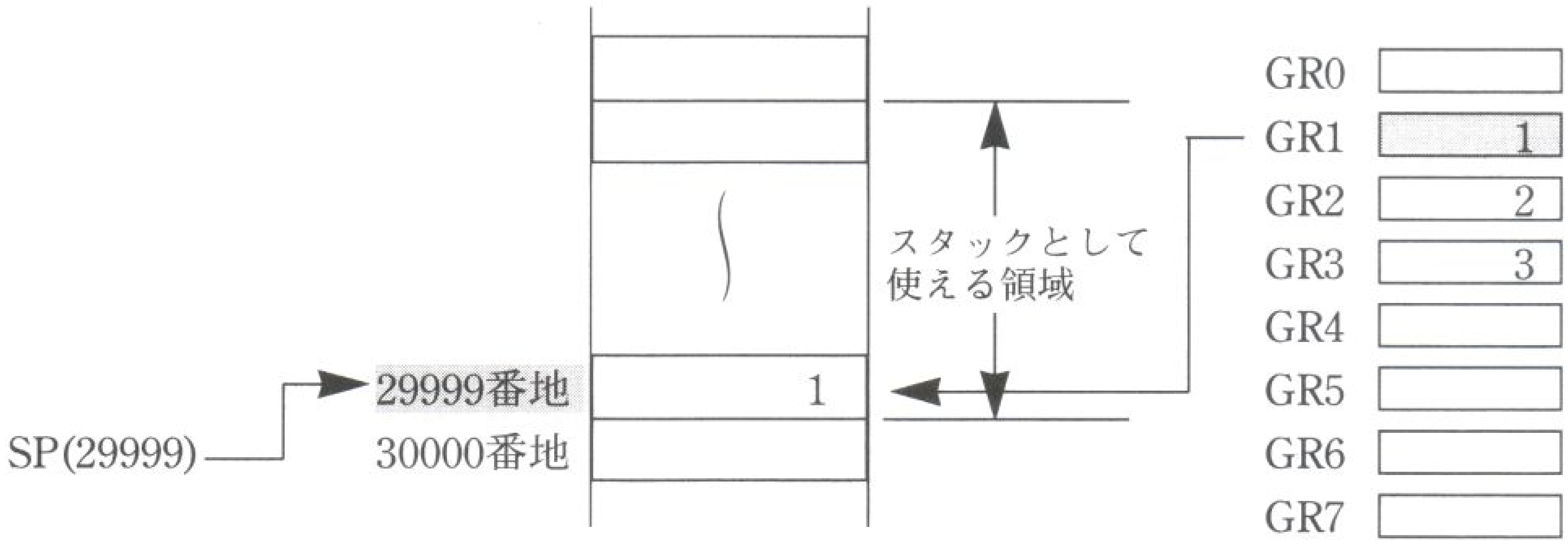
SPの値は、最初は30000(7530(16))番地になっています。




(*1) しかし、あまりに単純すぎて、「なんだこれは」と思う人もいるかもしれません。

このとき、最初の命令「PUSH 0,GR1」を実行すると、(1) SPの値を1つ引き算(論理減算)します。この例では29999番地になります。(2) 次に、レジスタGR1の内容を、SP(スタックポインタ)が示している主記憶装置(29999番地目)に格納します。この2段階の操作を前述のようにプッシュと呼びます。この様子を次に示します。

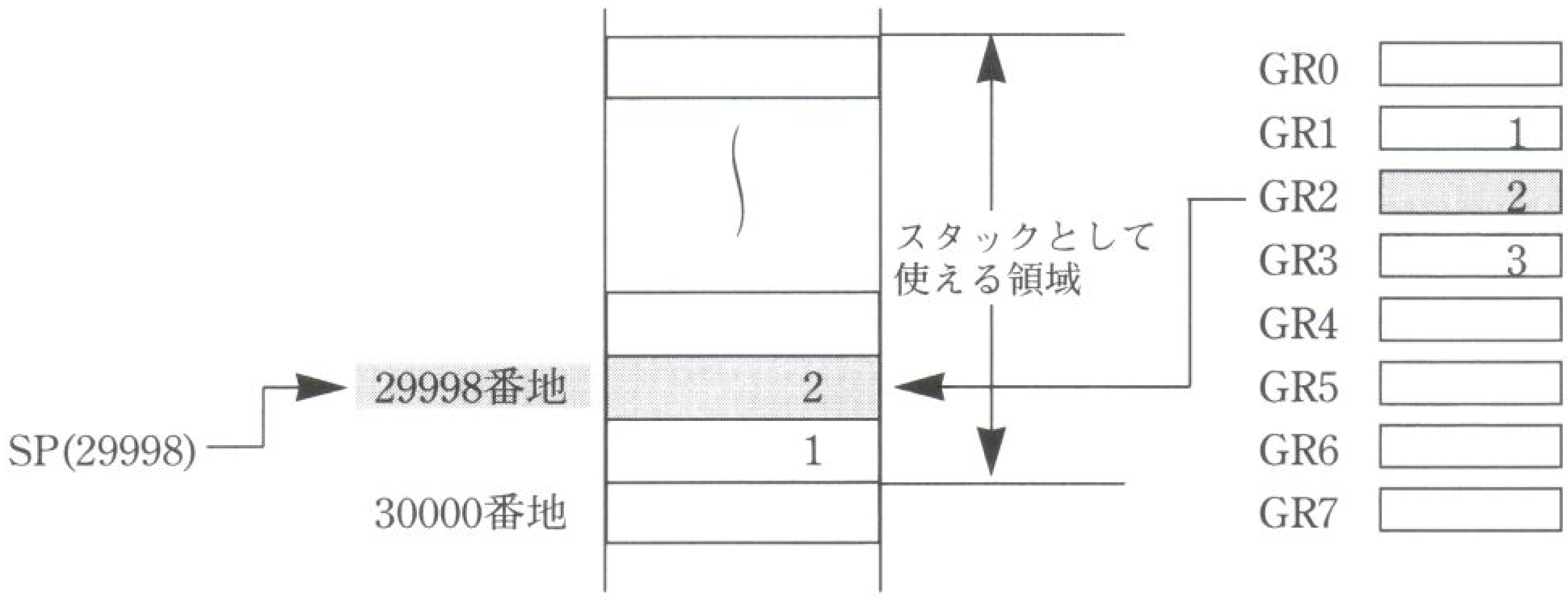
```
PUSH     0, GR1
```




この例から明らかなようにスタックが最初に指している番地(この例では3000番地)はスタックとしては使われません(＊1)。

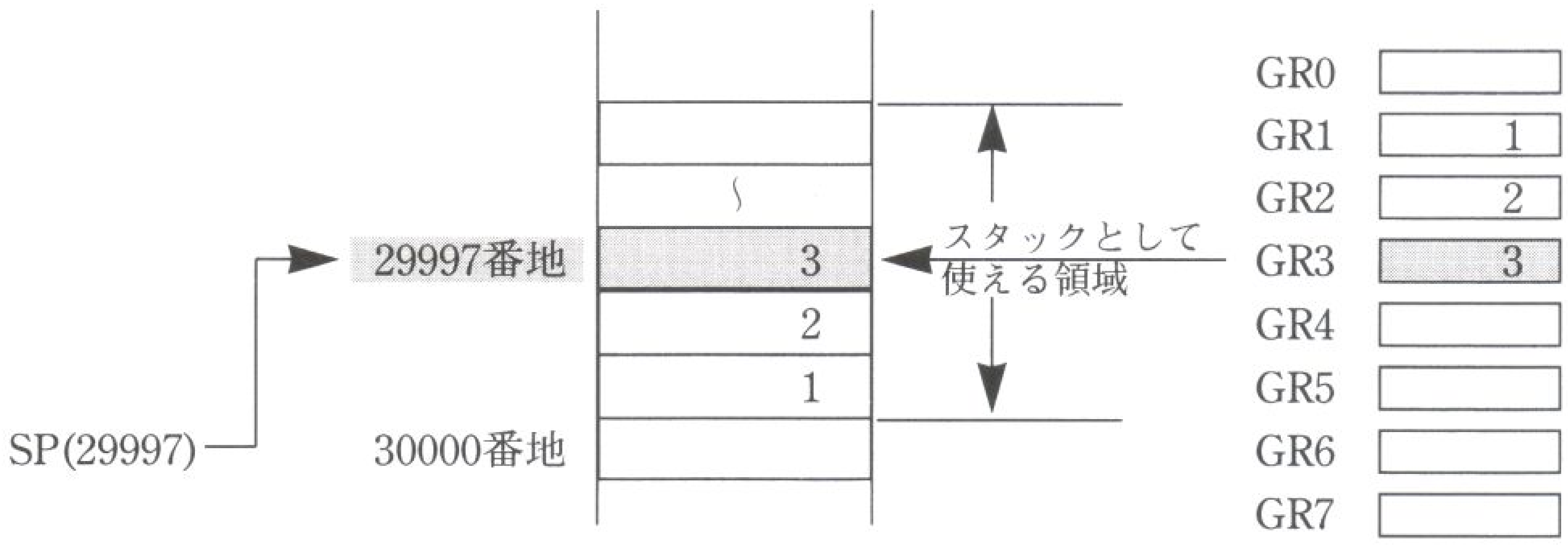
次に「PUSH 0,GR2」の処理を行うと次のようになります。

```
PUSH    0,GR2
```




さらに、「PUSH 0,GR3」の処理を行うと次のようになります。

```
PUSH    0,GR3
```

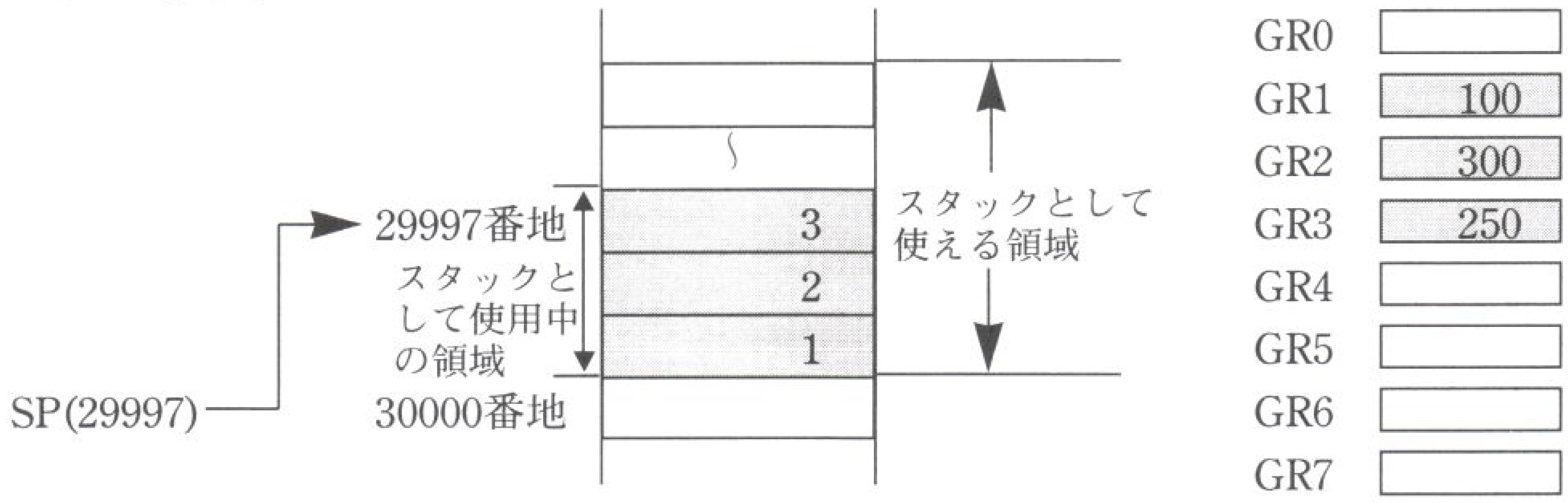



このように、PUSH命令によって汎用レジスタの値をスタック上に次々に積み上げることができま

(＊1) COMET IIの「仕様」からここには本プログラムがOSへ戻るときの戻り番地が格納されている可能性もありますが、CASLDVでは、OSに戻るRET命令は特別扱いでスタックを使用しないのでここ(30000番地)には何も入っていません。

す。

この状態で何らかのプログラムを実行して、レジスタGR1、GR2、GR3の内容が変わってしまったとしましょう。




以前の汎用レジスタの値が、スタック上に残されているので、これを元の汎用レジスタに戻すことを考えましょう。このスタックに積み上げられたデータを元の汎用レジスタに戻すには、POP命令を実行します。

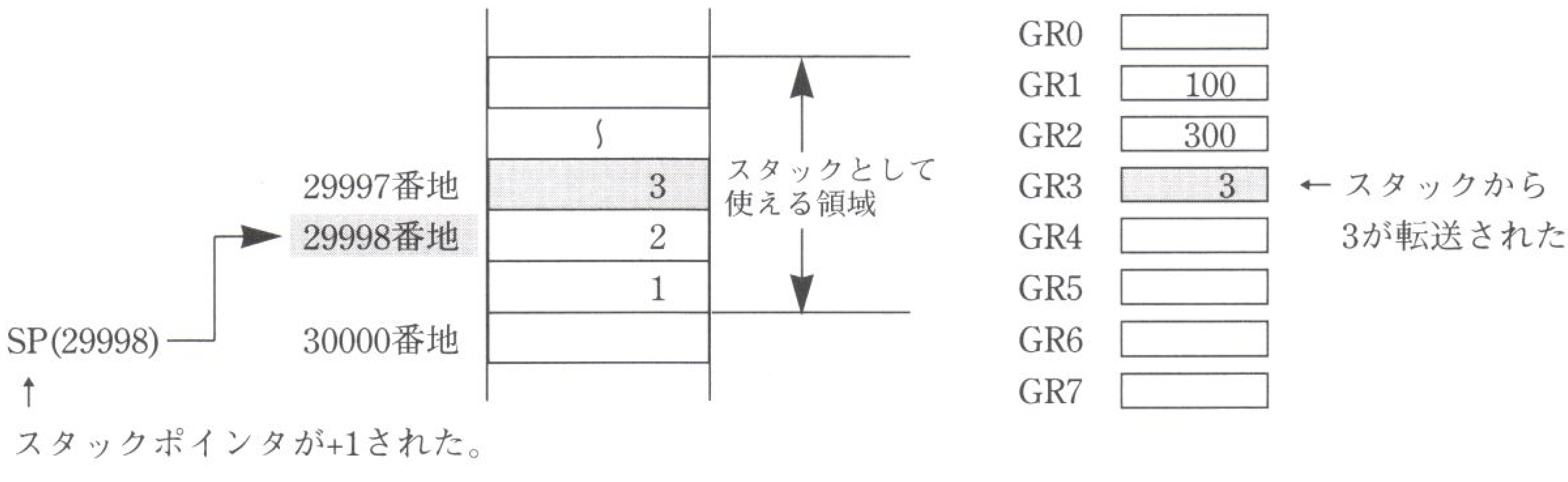
この場合には、次の命令で汎用レジスタの内容を以前の状態に復元できます。

```
POP  GR3
POP  GR2
POP  GR1
```

最初の「POP GR3」を実行するとどのようになるのでしょうか。「POP」命令を実行すると、次のステップが実行されます。(1) SPが指している記憶領域のデータ(この場合、29997番地目)をレジスタ(この例では、GR3)に転送します。(2) SPの値に1を加える(この場合、29998番地になります)。なお、アドレスデータは符号無しデータとして扱われるので、この＋1は論理加算(*1)となります。

さて、それでは、「POP GR3」が実行されると、次のようになります。

```
POP    GR3
```

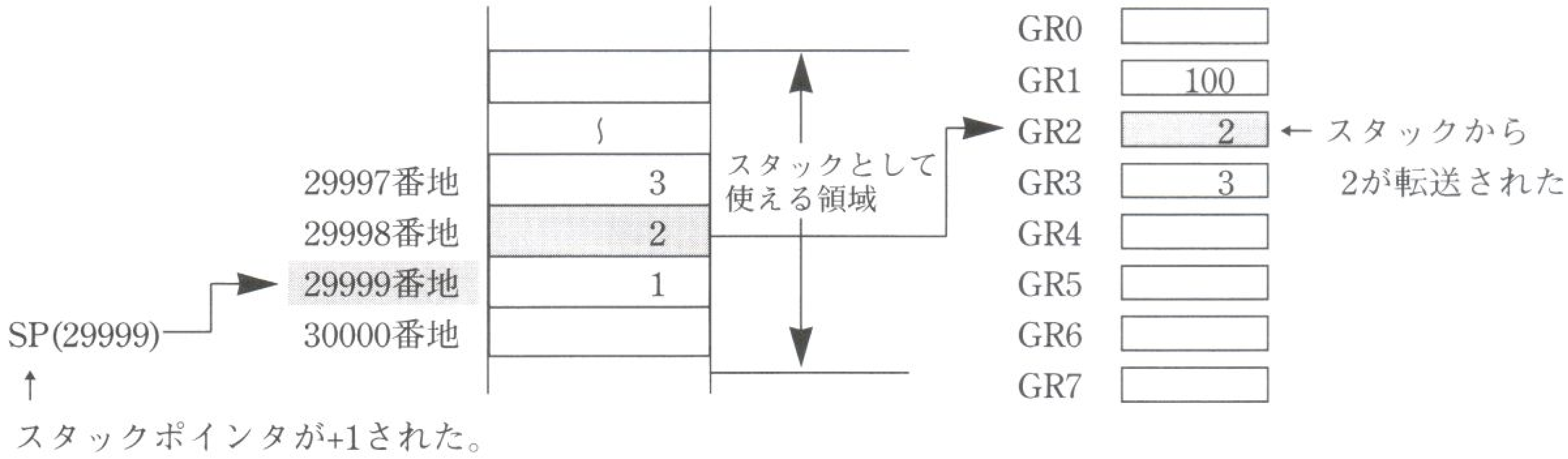



このとき、主記憶装置の29997番地目には値(3)が残っていますが、スタックポインタが指しているのは29998番地になります。スタック領域として、この時点で有効な領域は、29998番地～29999番地であるので、29997番地は、スタック領域として有効な値ではなくなります。たまたま、前の値が残っているに過ぎません。PUSH命令が実行されれば、別の値に置き換わります。

(*1) 論理加算でも算術加算でも計算される結果は同じであり、かつ、フラグレジスタはスタック操作では設定されないので、ここでわざわざ論理加算であると断らなくてもよいかも知れません。しかし、アドレスデータは、0～65535であるということを忘れると、場合によってはプログラムミスを引き起こす可能性もあるので、あえて記述しています。「仕様」中にも$+_L$と記述され、論理加算であることが明確に示されています。

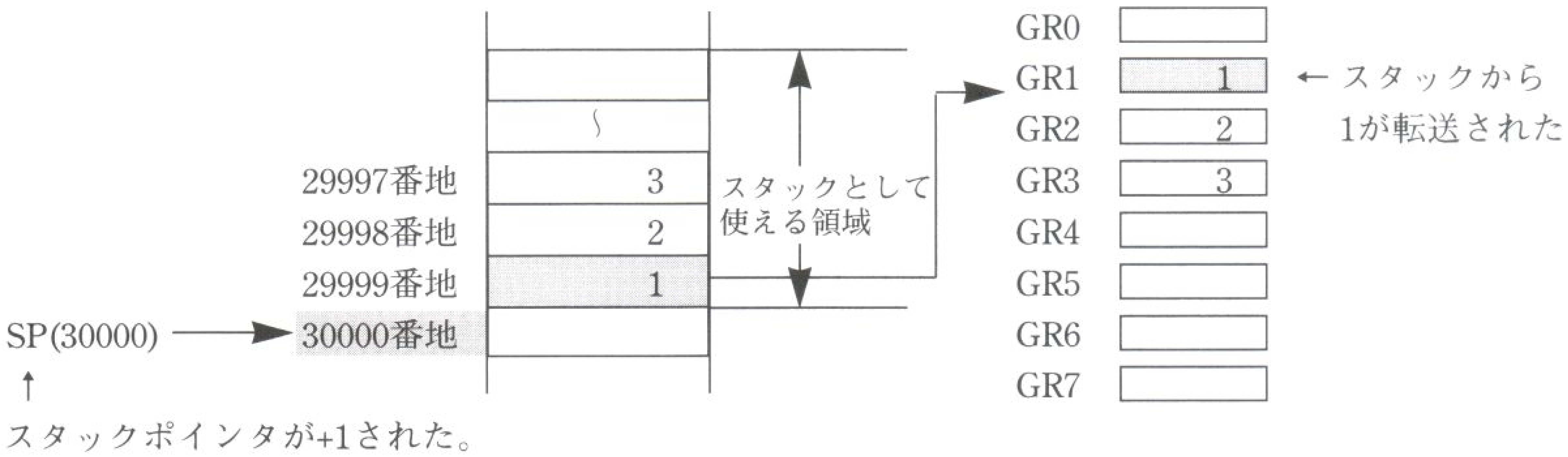
次に、「POP GR2」が実行されると次のようになります。

```
POP GR2
```




最後に、「POP GR1」が実行されると次のようになります。

```
POP GR1
```




これで、完全に汎用レジスタGR1〜GR3が元の値に復元されました。なお、現在SPは30000番地を指しており、この段階でさらにまたPOP命令を実行すると、それは、無意味なデータを汎用レジスタに転送するばかりか、最初に決めていたスタック領域の範囲の約束事を破ることになり、決して行ってはいけません。

しかし、COMET IIは、一般的には、この約束事を破ったことをチェックすることはできません(*1)。スタック領域を正しく使用するというのは、プログラムを書く人の責任になります。要するにスタックとは、そこにデータを積み上げ、そこからデータを取り出すというだけのものです。

ここで、紹介したように、スタックの簡単な使い方は次のように汎用レジスタの値の復元です。

```
PUSH  0,GR1
PUSH  0,GR2
PUSH  0,GR3
  ∫
レジスタの内容を変更するプログラム
  ∫
POP    GR3  元の値に戻す
POP    GR2
POP    GR1
```

(＊1) 本書に付属のCASLDVは初心者を対象としているのでこのチェックを行っています。

なお、実際にはあまり使いませんが、次のようにすれば、汎用レジスタの値にある値を加算したデータをスタックに積むことができます。

```
PUSH      10,GR1  GR1＋10の値をスタックに積む
PUSH      TBL,GR1「GR1＋TBLのアドレス」の値をスタックに積む
```

なお、前述のように、最初にSPの値であった30000番地にはデータが格納されません。つまり、最初のSPの値は、PUSHされたときに－1だけ論理減算されるので、そこには、データは格納されないのです。したがって、正確に述べるとスタック領域は、29999番地から始まるということになります。もちろん、余計にPOP命令を実行してしまうと、30000番地もスタックとして使われてしまいますが、この場合はプログラムのエラーです。

また、PUSH命令の一般形式からも分かるように、PUSH命令は次のような記述方法も可能です。

```
PUSH    ラベル
PUSH    定数
```

たとえば、次のようなプログラムを書くことができます。

```
        PUSH  100
        PUSH  LABEL
        ;何らかの処理
        POP   GR1
        POP   GR2

LABEL   DC    10
```

最初のPUSH命令(PUSH 100)では、定数100がスタックにプッシュされます。次の「PUSH LABEL」では、LABELの配置されているアドレスがプッシュされます。

## 6.8 コール、リターン命令

**コール、リターン命令**は、サブルーチンの処理に関する命令で、次のものがあります。

**表6-21 コール、リターン命令**

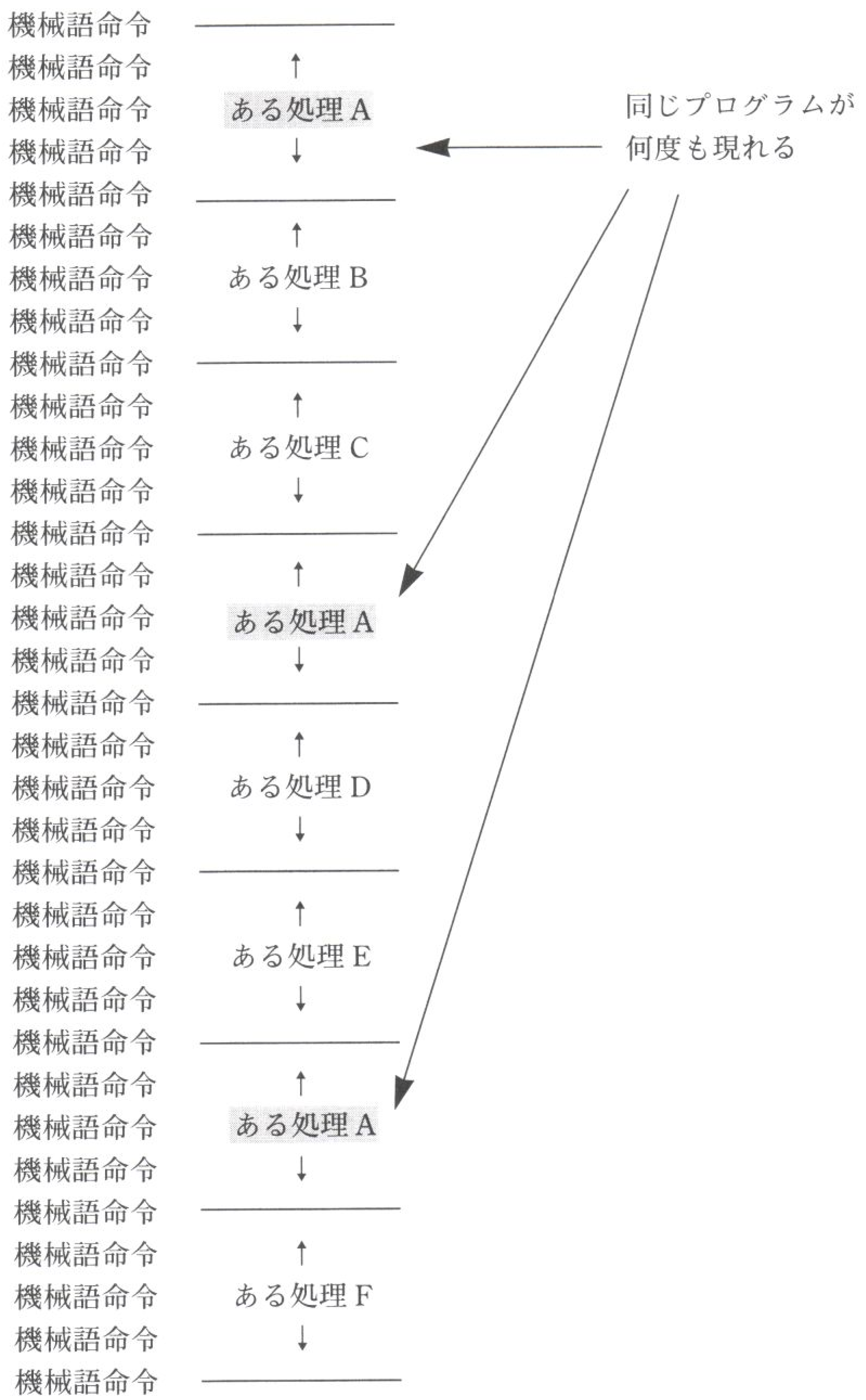
| 命　令 | 書き方<br>命令コード | <br>オペランド | 命令の説明 |
|---|---|---|---|
| コール<br>CALL subroutine | CALL | adr[,x] | SP ← (SP) $-_{L}$ 1<br>(SP) ← (PR)<br>PR ← 実効アドレス |
| リターン<br>RETurn from subroutine | RET | r | PR ← ( (SP) )<br>SP ← (SP) $+_{L}$ 1 |

さて、サブルーチンとは何なのでしょうか。**サブルーチン**(subroutine)とは、プログラムの実行の単位であり、CALL命令によって呼び出され、RET命令によって呼び出したプログラムに戻るという機能です(＊1)。しかし、この説明ではよく分からないと思いますので詳しく説明しましょう。一般に、プログラムとは、次に示すようにさまざまな処理の部分から成り立っています。

```
機械語命令  ──────────
機械語命令       ↑
機械語命令    ある処理 A
機械語命令       ↓
機械語命令  ──────────
機械語命令       ↑
機械語命令    ある処理 B
機械語命令       ↓
機械語命令  ──────────
機械語命令       ↑
機械語命令    ある処理 C
機械語命令       ↓
機械語命令  ──────────
機械語命令       ↑
機械語命令    ある処理 D
機械語命令       ↓
機械語命令  ──────────
機械語命令       ↑
機械語命令    ある処理 E
機械語命令       ↓
機械語命令  ──────────
機械語命令       ↑
機械語命令    ある処理 F
機械語命令       ↓
機械語命令  ──────────
```

(＊1) 第12章では、章の題材としてサブルーチンを取り上げています。

ところが、場合によっては、同じ処理が一つのプログラムの中に複数現れる場合があります。




たとえば、ちょっと複雑ですが、次の例を考えてみましょう。ラベルで定義された領域(A,B,C,D,E,F)の値を次の計算式に従って計算し、その値を求めるというものです。

```
Z ← A×B + C×D + E×F
```

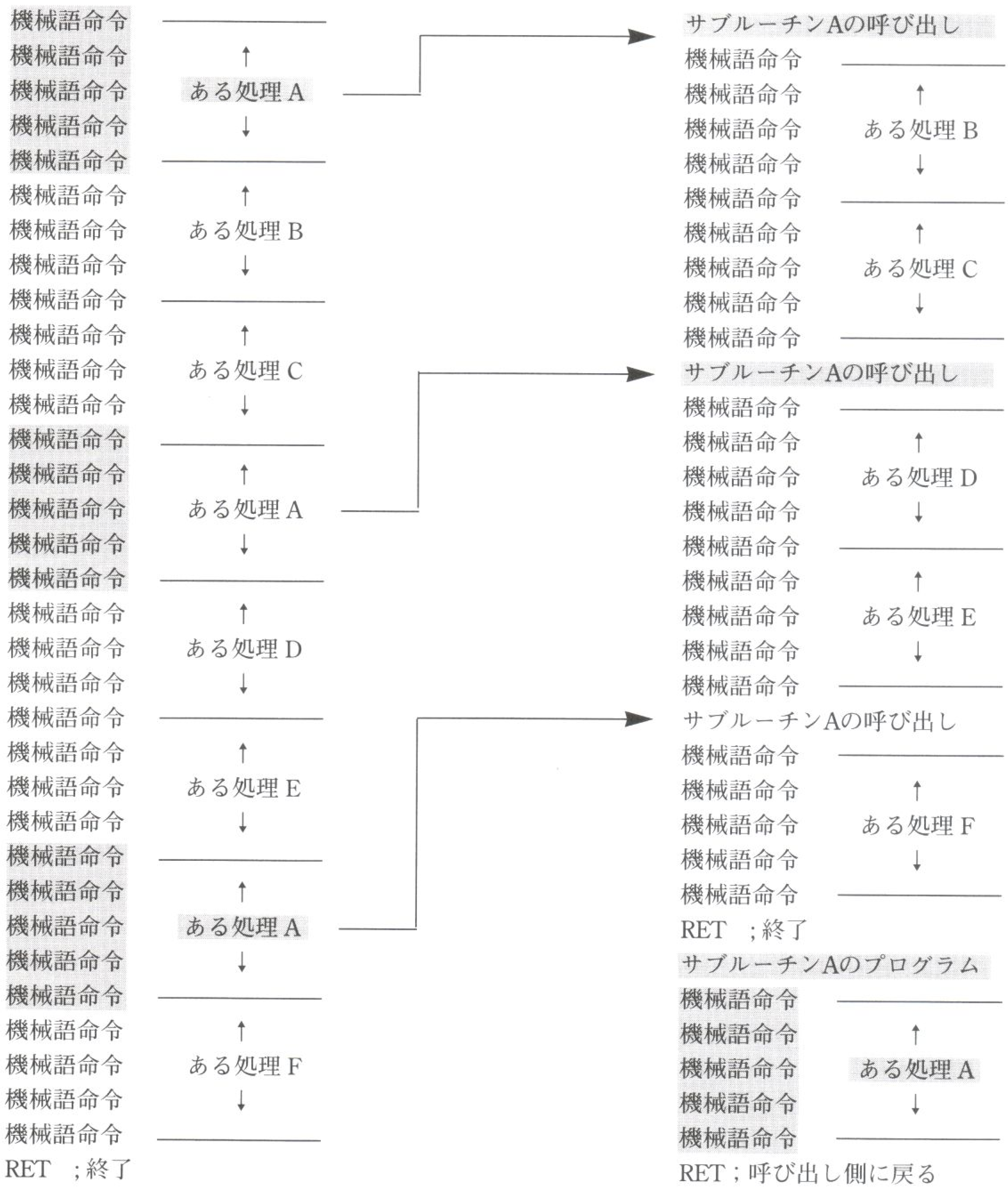
この計算式には、乗算が3回現れていますが、COMET IIには乗算の命令がありません。片方が定数なら(たとえば、A×2やA×10)ならば、シフト演算などを組み合わせて効率よくプログラムが組めますが、A×Bのように任意のデータとなるとちょっと面倒です(＊1)。

このプログラムの骨格は次のようになるでしょう。

(＊1) 実際の乗算プログラムは、9.4.1項、10.6節、12.3節で説明します。特に12.3節ではここで説明している方法でプログラムを作成しています。

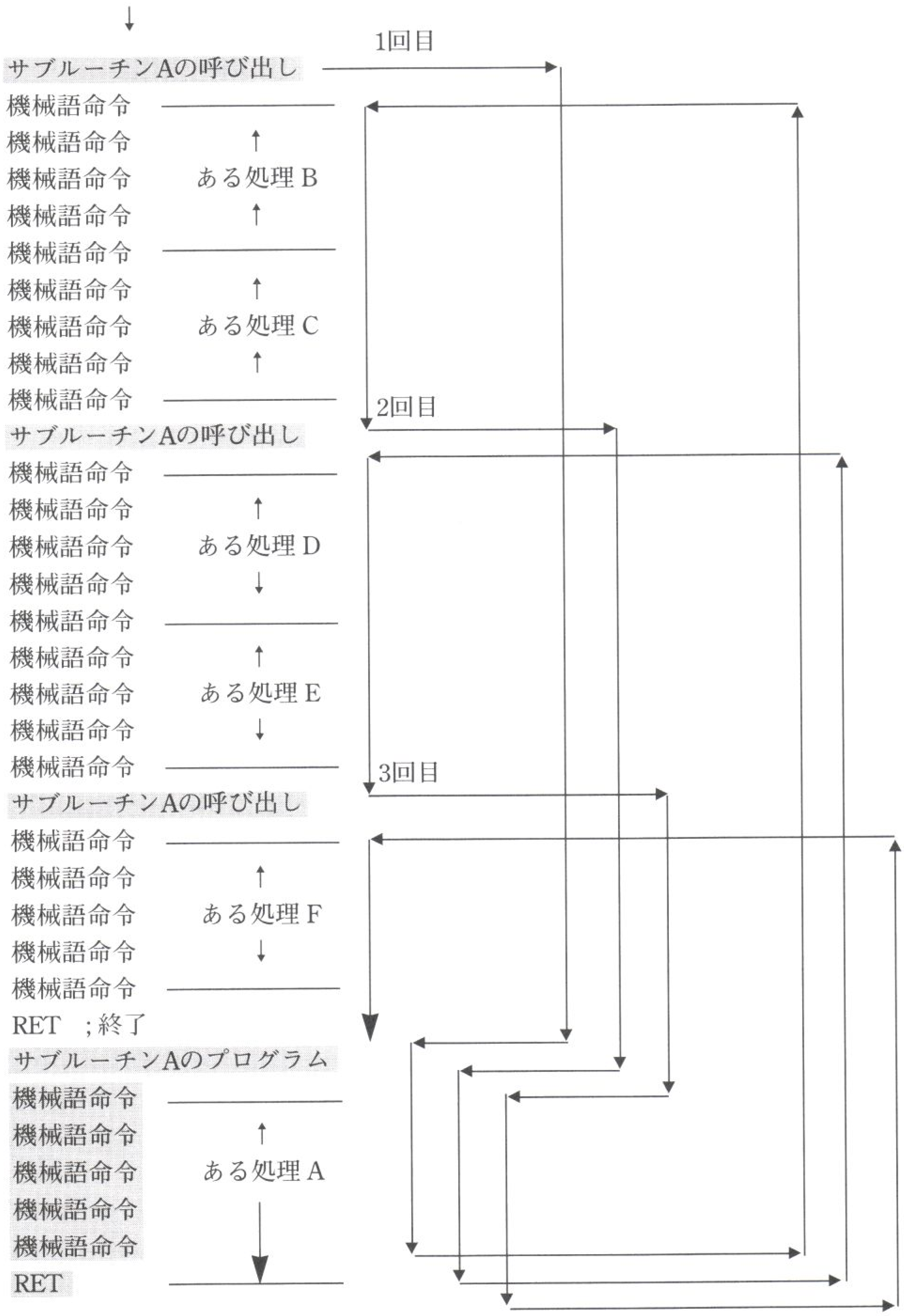
```
GR0 ← 0
A×Bを計算して、GR0に加える
C×Dを計算して、GR0に加える
E×Fを計算して、GR0に加える
GR0 → Z
```

この処理には、乗算を行うというプログラムが3回現れます。そこで、通常は、このようなプログラムを、サブルーチンとして別の場所に記述し、サブルーチン呼び出し(コール)という方法で、その部分を実行します。先に示したプログラムでは、次のようになります。




サブルーチンを使わない形式　　　　サブルーチンを使う形式

このプログラムの実際の動作は次のようになります。



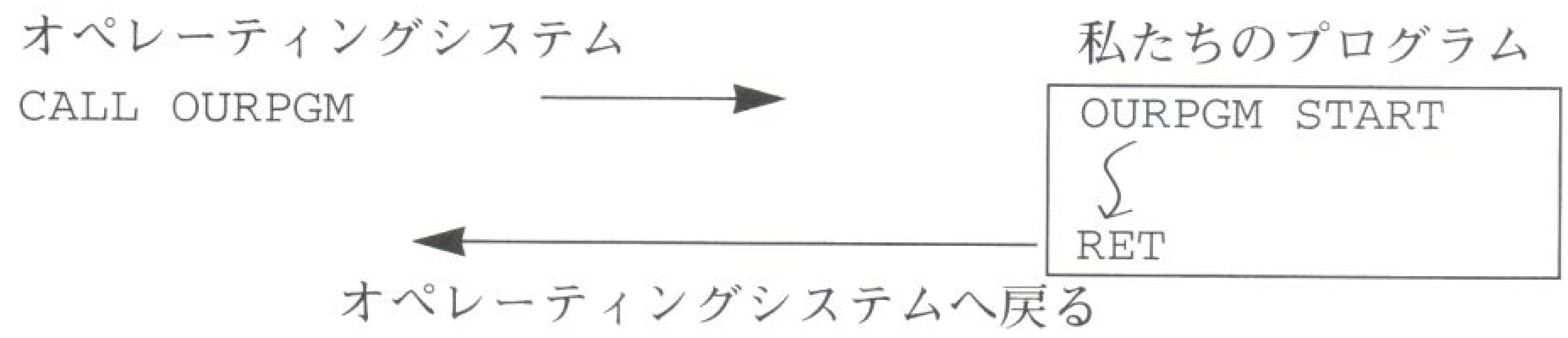

サブルーチンを呼び出すには、CALL命令を使います。

```
CALL    1000    1000番地から始まるサブルーチンを実行する
CALL    SUB     ラベルSUBで始まるサブルーチンを実行する
```

なお、サブルーチンの中から、呼び出し側に戻るにはRET命令を使います。

```
RET
```

プログラムを終了させる命令もRETでした。私たちが今まで実行してきたプログラムも実は、オペレーティングシステムから呼び出されていたサブルーチンであったのです。そこで、オペレーティングシステムへ戻るためにRET命令を用いていたのです。




さて、それでは、簡単なCASL II のプログラムを紹介しましょう。これは、レジスタのGR1とGR2をゼロにするR12CLEARというサブルーチンを3回コールしているプログラムです(＊1)。

```
   Machine        Source List (subsamp.casl)
Adr. Code       Line  Program
                  1  SUBSAMP   START
  0   8000        2            CALL   R12CLEAR
  1   000D
  2   1210        3            LAD    GR1, 10
  3   000A
  4   1220        4            LAD    GR2, 15
  5   000F
  6   8000        5            CALL   R12CLEAR
  7   000D
  8   1210        6            LAD    GR1, 20
  9   0014
  A   8000        7            CALL   R12CLEAR
  B   000D
  C   8100        8            RET    ;プログラムの終了：オペレーティングシステムに戻る
  D   2511        9  R12CLEAR  SUBA   GR1, GR1
  E   2522       10            SUBA   GR2, GR2
  F   8100       11            RET    ;サブルーチンの終了：呼び出し側に戻る
                 12            END
```

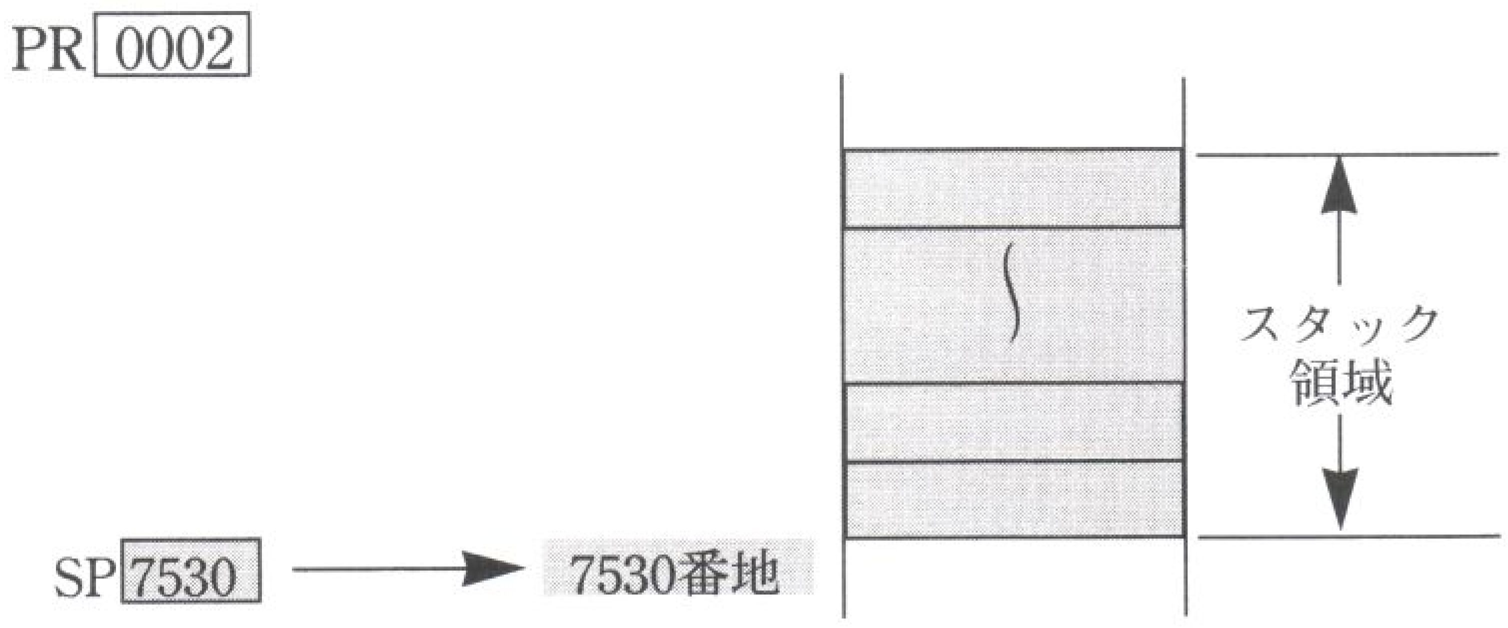
このプログラムを0番地にロードして実行した結果を次に示します。

(＊1) このプログラムでは、汎用レジスタの値をゼロクリアするのに、自分の値から自分の値を引くことによって実現しています。しかし、実際には「XOR　GR1,GR1」とXOR命令を使う方が効率的です(7.1.2項参照)。

```
PR   Instruction       SP   OSZ  GR0  GR1  GR2   Explanation
0000 CALL  000D        752F ---  8029 C823 98BE
000D SUBA  GR1,GR1     752F --Z  8029 0000 98BE  GR1:[s:0] -=> GR1:[s:0]
000E SUBA  GR2,GR2     752F --Z  8029 0000 0000  GR2:[s:0] -=> GR2:[s:0]
000F RET               7530 --Z  8029 0000 0000  Return to 0002
0002 LAD   GR1,000A    7530 --Z  8029 000A 0000  [h:000A, s:10, u:10]
0004 LAD   GR2,000F    7530 --Z  8029 000A 000F  [h:000F, s:15, u:15]
0006 CALL  000D        752F --Z  8029 000A 000F
000D SUBA  GR1,GR1     752F --Z  8029 0000 000F  GR1:[s:0] -=> GR1:[s:0]
000E SUBA  GR2,GR2     752F --Z  8029 0000 0000  GR2:[s:0] -=> GR2:[s:0]
000F RET               7530 --Z  8029 0000 0000  Return to 0008
0008 LAD   GR1,0014    7530 --Z  8029 0014 0000  [h:0014, s:20, u:20]
000A CALL  000D        752F --Z  8029 0014 0000
000D SUBA  GR1,GR1     752F --Z  8029 0000 0000  GR1:[s:0] -=> GR1:[s:0]
000E SUBA  GR2,GR2     752F --Z  8029 0000 0000  GR2:[s:0] -=> GR2:[s:0]
000F RET               7530 --Z  8029 0000 0000  Return to 000C
000C RET               7530 --Z  8029 0000 0000  Return to OS
```

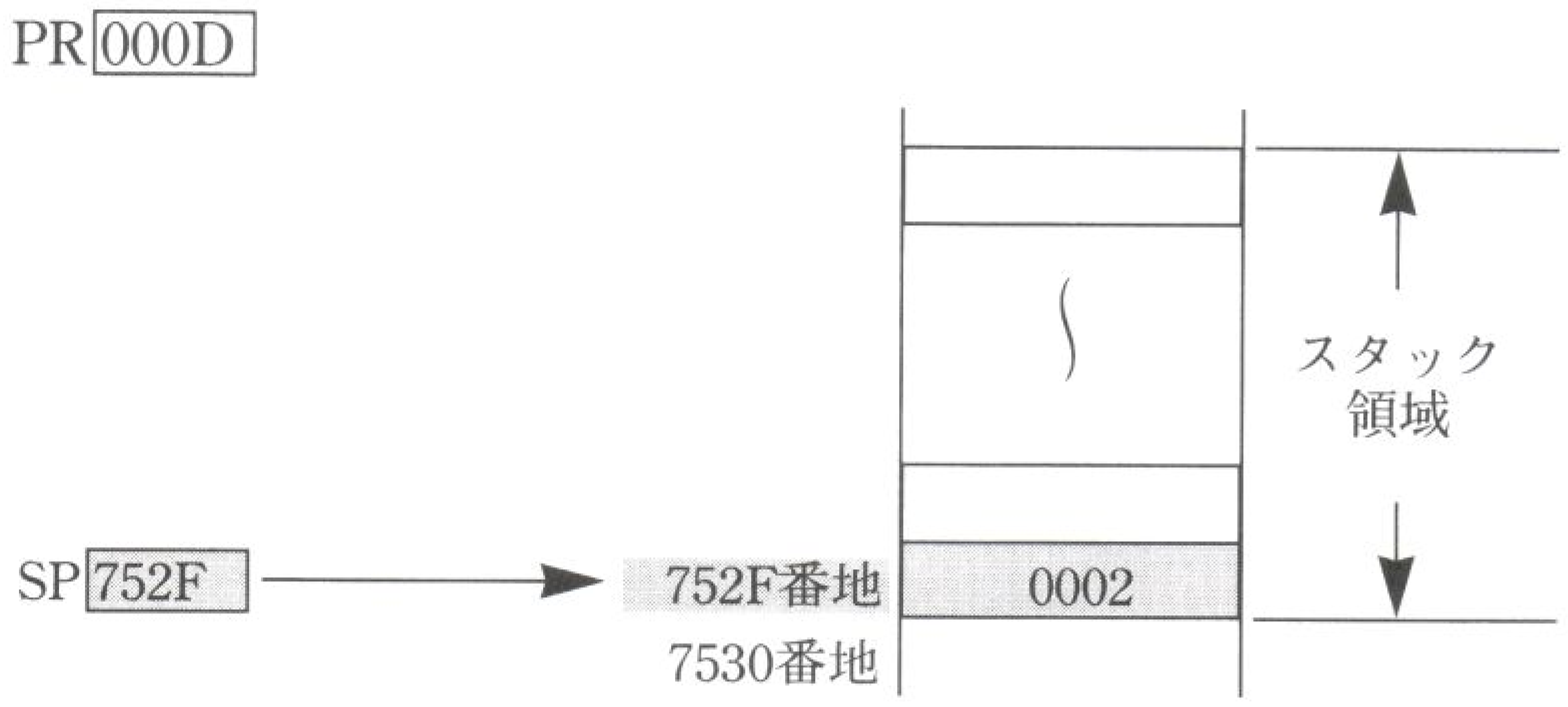
CALL命令とRET命令を実行すると、スタックポインタが変化しています(＊1)。実は、サブルーチン呼び出しは、スタックの機能を利用して実現しています。

プログラムの1行目の「CALL R12CLEAR」の実行直前、COMET IIの内部は次のようになっています。ただし、スタックポインタのアドレスは、16進法で記述しています。PRは、「CALL R12CLEAR」の次に実行すべきアドレス(2番地)を保持しています。




「CALL　R12CLEAR」を実行すると、これは、次の操作を行います。(1) SPを一つ減じる。この例では、$7530_{(16)}-1=752F_{(16)}$になります。(2) PRレジスタの値(CALL R12CLEAR命令の次の命令のアドレス、この場合2番地)をそのSPが指している部分に転送します。この2番地はサブルーチンから戻ってきたときに実行すべきアドレスなので、**戻り番地**と呼びます。つまり、(1),(2)は、「CALL R12CLEAR」命令の次のアドレスをスタックにPUSHしているのです。そして、(3) CALL命令の実効アドレス(この場合、R12CLEARラベル、すなわち$D_{(16)}$番地)をPRに設定します。すると、次の実行が$D_{(16)}$番地になります。

(＊1) プログラムを終了させるRETを実行したときには、スタックポインタの値が変化していません。これはCASLDVがプログラムを終了させる(OSに戻る)RETを特別に扱っているためです。




サブルーチンの内部を実行し、RET命令を実行すると、次のステップが実行されます。

(1) SPの指しているスタック上の値(戻り番地)をPRにセットする。
(2) SPの値を1つ増やす。

つまり、スタック上のデータ(戻り番地)をPOPして、PRにセットしているだけなのです。このようにすると、次に実行すべき命令は、呼び出した「CALL R12CLEAR」命令の次の命令になります。そこで、この例では、サブルーチンから戻ってきたときに2番地目の「LAD　GR1, 10」が実行されるのです。

なお、実際にサブルーチンを使うには、サブルーチンへのデータの渡し方など、さまざまな知識が必要になりますが、それらについては、第12章で扱います。

## 6.9 その他の命令

その他の命令として、次のものがあります。

表6-22 その他の命令

| 命　令 | 書き方 | | 命令の説明 |
|---|---|---|---|
| | 命令コード | オペランド | |
| スーパバイザコール<br>SuperVisor Call | SVC | adr[,x] | 実効アドレスを引数として割り出しを行う。<br>実行後の汎用レジスタGRとフラグレジスタFRの内容は不定になる。 |
| ノーオペレーション<br>No OPeration | NOP | | 何もしない。 |

### 6.9.1 SVC命令

**スーパバイザコール**とは、オペレーティングシステムの機能を利用するための命令です。通常、オペレーティングシステムは、そのシステムの上で動作するプログラムを書く人のために、さまざまなサブルーチンを用意しています。さまざまな理由で、それらをCALL命令で呼び出すことができない場合があります。そこで、この命令を使って、オペレーティングシステムに、さまざまな処理を依頼します。

どのような処理が用意されているのか、残念ながら情報処理技術者試験案内の「仕様」には、書かれていません。参考資料の中で、「SVC　1」がデータの入力であることが示されていますが、これ以外の情報は何もありません。

本書に付属のCASLDVは、次のスーパバイザーコールを提供しています。

表6-23 本書に付属のCASLDVが提供するスーパバイザコール

| 番号 | 説　明 | パラメタ |
|---|---|---|
| 1 | 文字データの入力 | GR1に入力バッファアドレス、GR2に文字数取得エリアアドレス |
| 2 | 文字データの出力 | GR1に出力バッファアドレス、GR2に出力文字数 |

### 6.9.2 NOP命令

NOP命令とは、何もしない命令です。プログラムの中でアドレスをずらしたくない時の穴うめや実行のタイミングを取る時に使います。

たとえば、次のプログラムを見てください。

```
  Machine       Source List (NOPTI1.casl)
  Adr.Code    Line     Program
                 1  NOPTI1    START
  0  1010        2            LD      GR1,#0009        GR1 ← 3
  1  0009
  2  2010        3            ADDA    GR1,#000A        GR1 ← GR1 ＋ 5
  3  000A
  4  2010        4            ADDA    GR1,#000A        GR1 ← GR1 ＋ 5
  5  000A
  6  1110        5            ST      GR1,#000B        GR1 → B(16)番地
  7  000B
  8  8100        6            RET
  9  0003        7            DC      3
  A  0005        8            DC      5
  B  0000        9            DS      1
                10            END
```

これは、3＋5＋5＝13を求めて、B(16)番地に格納するプログラムです。アドレスがすべて絶対アドレス(9番地、A(16)番地、B(16)番地)になっているので、このプログラムは0番地からロードしないと正しく動きません。さて、プログラムを変更して、3＋5＝8を求めるものに変えたいと思います。4行目の「ADDA GR1,#000A」を取ればよいと思われます。しかし、次のようにアドレスがずれて、データの番地は7番地、8番地、9番地になってしまいます。

```
  Machine        Source List (NOPTI2.casl)
  Adr.Code    Line     Program
                 1  NOPTI2    START
  0  1010        2            LD      GR1,#0009        GR1 ← 3
  1  0009
  2  2010        3            ADDA    GR1,#000A        GR1 ← GR1 ＋ 5
  3  000A
  4  1110        4            ST      GR1,#000B        GR1 → B(16)番地
  5  000B
  6  8100        5            RET
  7  0003        6            DC      3
  8  0005        7            DC      5
  9  0000        8            DS      1
                 9            END
```

前述のように、3が格納されているアドレスは7番地になっています。これを強引に実行すると、次のようにまったく異なる値が計算されます。次の例では、たまたま9番地～B(16)番地の値がゼロだったようです。

```
PR   Instruction        SP   OSZ  GR0  GR1   Explanation
0000 LD    GR1,0009     7530 --Z  8029 0000  0009:[h:0000, s:0, u:0]
0002 ADDA  GR1,000A     7530 --Z  8029 0000  000A:[h:0000,s:0]+=> GR1:[s:0]
0004 ST    GR1,000B     7530 --Z  8029 0000  GR1:[h:0000, s:0, u:0]
0006 RET                7530 --Z  8029 0000  Return to OS
```

そこで、このような場合、NOP命令で穴埋めします。NOP命令の語長は1語で、「ADDA GR1,#000A」命令の語長は2語ですから2つNOP命令が必要です。

```
  Machine       Source List (NOPTI3.casl)
 Adr.  Code   Line     Program
     |      |   1 | NOPTI3    START
   0 | 1010 |   2 |           LD     GR1,#9    GR1 ← 3
   1 | 0009 |     |
   2 | 2010 |   3 |           ADDA   GR1,#A    GR1 ← GR1 + 5
   3 | 000A |     |
   4 | 0000 |   4 |           NOP
   5 | 0000 |   5 |           NOP
   6 | 1110 |   6 |           ST     GR1,#B    GR1 → B(16)番地
   7 | 000B |     |
   8 | 8100 |   7 |           RET
   9 | 0003 |   8 |           DC     3
   A | 0005 |   9 |           DC     5
   B | 0000 |  10 |           DS     1
     |      |  11 |           END
```

3が格納されているアドレスが9番地に戻っています。このプログラムを実行すると、正しく動作していることが分かります。これで、ちゃんと8が求められました。

```
PR   Instruction        SP   OSZ  GR0  GR1   Explanation
0000 LD    GR1,0009   7530  ---  8029 0003  0009:[h:0003, s:3, u:3]
0002 ADDA  GR1,000A   7530  ---  8029 0008  000A:[h:0005,s:5]+=> GR1:[s:8]
0004 NOP              7530  ---  8029 0008
0005 NOP              7530  ---  8029 0008
0006 ST    GR1,000B   7530  ---  8029 0008  GR1:[h:0008, s:8, u:8]
0008 RET              7530  ---  8029 0008  Return to OS
```

*Memorandum*

# 第7章 直線型のプログラミング

本章では、CASL IIプログラミングの基本的かつ実践的なテクニックを学びます。基本的なことといっても、本章で学習する内容には、COMET IIの本質的な事柄も含まれていますので、しっかり学習してください。また、本章では、第6章で個別に学習したさまざまな命令を有機的に関連付けて説明しますので、個々の機械語命令の特徴などが理解できると思います。情報処理技術者試験などで問題を解く際には、個々の命令の特徴などをしっかりと理解しておくことが必須条件です。特にCOMET IIでは機械語の機能が増え、同じ処理を多くの異なる機械語命令で書けてしまいます。そこで、近年のレベルの高い問題に対しては、他の命令との違いを認識しておくことによって、誤解答することを避けることができます。

## 7.1 汎用レジスタへの定数の設定

本節では、汎用レジスタに定数を設定するさまざまな方法を説明します。7.1.1項で「任意の定数」を設定する方法について説明し、7.1.2項で「ゼロ」を設定する方法について説明します。ゼロを汎用レジスタに設定するという処理は非常に多くの場面で使う上に処理が少し特殊になります。そこで、この方法は特別に扱うことにします。

### 7.1.1 汎用レジスタへの任意の定数の設定

COMET IIの機械語を用いて、さまざまな方法で汎用レジスタに定数をセットすることができます。その方法を表7-1にまとめます。

表7-1 汎用レジスタGR1へ定数10を設定するさまざまな方法(*1)

| プログラム | 説　明 | 命令語の語数 | データ(10)の格納場所 |
|---|---|---|---|
| LAD GR1,10 | LAD命令を用いる(10は実効アドレス) | 2 | 不要 |
| LD GR1,TEN | ラベルTENの内容(10が入っている)をロードする | 2 | 要(1語) |
| LD GR1,=10 | リテラルを使う | 2 | 要(1語) |

汎用レジスタに値を設定するもっとも効率の良い方法は、LAD命令を使うことです。

```
LAD   GR1,10
```

LAD命令の一般形式は、LAD r,adr[,x]でした。上記の記述「LAD　GR1,10」は、ちょうどrの部分がGR1、adrの部分が10([,x]は省略)となって、adr[,x]によって生成される実効アドレスが10となりま

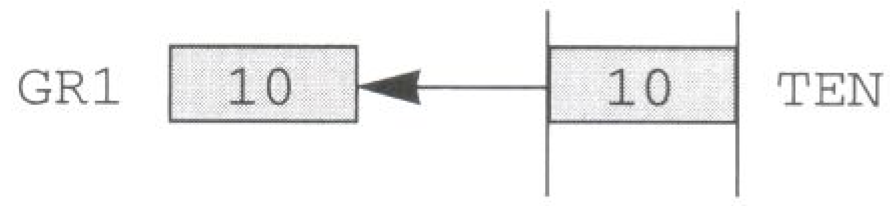
(*1) 何度もくり返しますが、COMET IIの仕様では、各命令の命令語長はまったく示されていません。しかし、参考資料の中で、機械語命令のビットパターンと共に、命令語長が示されており、本書では、そこに記述されている内容に完全に準拠しています。

すので、この値10がGR1にロードされるわけです(復習)。

一方、次の方法でも汎用レジスタに10をロードできます。

```
          LD       GR1,TEN

TEN       DC       10
```

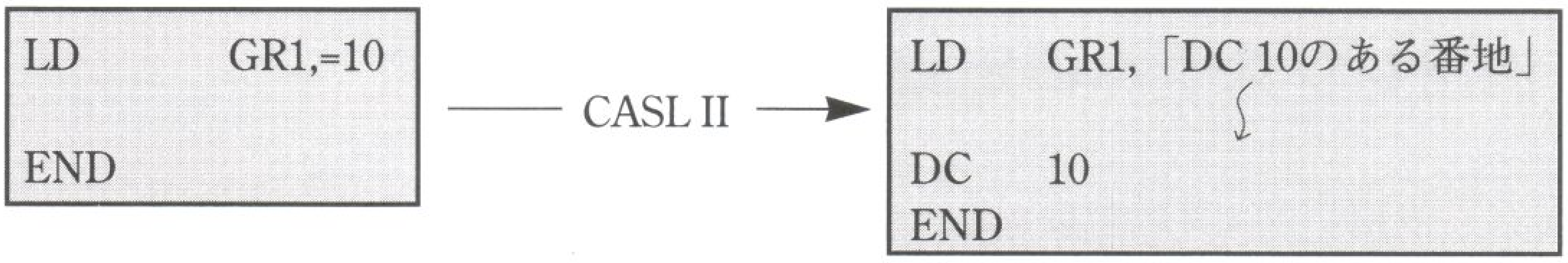



これは、TENというラベルの領域に10を確保し、「LD GR1,TEN」で実効アドレスTENの内容(値)をGR1にロードします(復習)。しかし、2個所(LD命令とDC命令)に渡って、プログラムを記述しなければならず、ちょっと面倒です。

一方、リテラルを使えば、次のように一行で書くこともできます。

```
          LD       GR1,=10
```

「＝10」の部分がリテラルになります。これは、LAD命令と同じように記述が1行ですみますので、プログラムを記述する立場から言えば楽です。しかし、リテラルは、COMET IIの機械語命令にはありません。リテラルを含む記述は、CASL IIアセンブラによって、次のように自動的に変換されます。「DC 10」(主記憶装置に10を確保する命令)は、END命令の直前に置かれます。




したがって、リテラルは、プログラムを書く際には楽なのですが、実際に生成される機械語は、ラベルを用いた記述と違いはありません。このことを知るために、次のプログラムを調べましょう。

```
GRSET     START
          LAD      GR1,10
          LD       GR1,TEN
          LD       GR1,=10
          RET
TEN       DC       10
          END
```

アドレスと生成された機械語を調べるために、アセンブラリストを示します。

```
  Machine     Source List (GRSET.casl)
  Adr.Code    Line     Program
            |   1 | GRSET     START
   0 | 1210 |   2 |           LAD    GR1,10
   1 | 000A |     |
   2 | 1010 |   3 |           LD     GR1,TEN
   3 | 0007 |     |
   4 | 1010 |   4 |           LD     GR1,=10
   5 | 0008 |     |
   6 | 8100 |   5 |           RET
   7 | 000A |   6 | TEN       DC     10
   8 | 000A |     | (リテラルによって生成)
            |   7 |           END
```

LAD命令も、LD命令も命令語長は2語になっています。また、リテラルを使った4行めのLD命令は、8番地の値(そこには10(=A(16))がセットされている)をロードするLD命令(*1)として機械語にアセンブルされていることがわかります。

さて、それでは、「定数を汎用レジスタにセットする」という意味で使った場合、LAD命令とLD命令は、どこが異なるのでしょうか。まず、この2つの機械語を比較しましょう。

```
LAD     GR1,10      1210 000A

LD      GR1,TEN     1010 0007
```

LAD命令は、その機械語の中に10(=000A(16))という数値が含まれています。このため、CPUが機械語をCPUの内部に取り込んだときに、すでに10という数値もCPUの中に取り込まれている可能性が高いのです(*2)。その場合には、すぐに10を汎用レジスタGR1に転送できます。

一方、LD命令の機械語の中には、10という数値が格納されているアドレス(この例では0007)が含まれています。そのため、CPUは、この機械語を解読した後、さらに、記憶装置にアクセスし、7番地目から値10を取り出し、それをCPU内の汎用レジスタGR1にロードするのです(*3)。

このように、実行効率の面から見ると、LAD命令に軍配があがります。それでは、「常にLAD命令を使えば良い」と思うかもしれません。しかし、LAD命令は、フラグレジスタを変更しないのです。

そのために、ロードした値によって処理を分岐するプログラムを書くことができません。これは、LAD命令の致命的な部分です(が、6.1.3項の注釈でも述べているように、試験問題との絡みでこのような設計になっていると思われます)。ロードした値によって処理を分岐するプログラムを書く場合は、LD命令を使います(*4)。

また、この他にも、LAD命令で定数を設定するよりもLD命令で設定した方が便利なことがあります。それは、同じ数値を何度もロードする場合です。1234という数値を何度もいろいろな汎用レジスタに設定する必要があることを考えましょう。

```
LAD     GR1,1234
            :
LAD     GR5,1234
            :
LAD     GR3,1234
            :
LAD     GR1,1234
            :
LAD     GR4,1234
            :
```

プログラム1

```
          LD      GR1,CONST
                      :
          LD      GR5,CONST
                      :
          LD      GR3,CONST
                      :
          LD      GR1,CONST
                      :
          LD      GR4,CONST
                      :
CONST     DC      1234
```

プログラム2

(*1) アセンブラリストを見ると、「LD GR1,=10」は「1010 0008」という機械語に変換されています。「1010」が「LD GR1」に相当し、「0008」は8番地目のデータが処理対象となることを示しています。8番目には、リテラルによって生成された値(10=000A(16))が格納されています。

(*2) COMET IIの命令の中で語長が2語の命令の第2語目は、どの命令もすべてアドレスです。ひょっとしたら、COMET IIはCPUには常に命令の第1語目だけをロードして解析をして、次にその第2語に格納されたアドレスを取り出す処理を行うかもしれません。実在のコンピュータが実在しないので、断定のしようがないのです。しかし、主記憶装置へのアクセスの回数は、LAD命令の方がLD命令より少ないことは明らかです。

(*3) ただし、ここに書いてある内容もまた、COMET IIの仕様にはっきり書いてあるわけではありません。しかし、機械語の世界の常識から、ここに書いてあることが予想できるのです。

(*4) ただし、ここでは定数の設定という限られた用途で用いることを前提に議論しています。設定された定数にって処理を分岐させるというプログラムは実際には書くことは少ないでしょう。

この2つのプログラムの違いは何でしょうか。実行の効率だけを考えれば、LAD命令の方が有利です。プログラムの占める記憶域の大きさ(プログラムの大きさ)もLAD命令の方が、CONSTの領域(DC 1234の部分)の無い分、少ないです(といってもたかだか1語分ですが)。

さて、ここで、この「1234」という定数を4321に変更することを考えてみましょう。すると、LAD命令の方は、プログラム中の5個所を変更しなければなりませんが、LD命令では、1個所だけを変更すればよいので、変更が大変楽です。

この例のように、同じ数値を、何度も利用する場合で、その値が頻繁に変更される可能性のある場合には、LD命令の方が良いでしょう。しかし、汎用レジスタに余裕があれば、次のようにした方が効率がさらに良いことはいうまでもありません。次の例では、レジスタGR7を値1234の保持用に用いています。「LD r1,r2」の形式の命令(たとえば、LD GR1,GR7)は、その命令の占める語が1語であり、さらに汎用レジスタの値の複写なのでCPU内だけのデータ転送で処理が完了する命令であり、主記憶装置とのアクセスが発生しないという利点があります。

```
        LD      GR7,CONST     GR7 ← CONST
        LD      GR1,GR7
                    :
        LD      GR5,GR7
                    :
        LD      GR3,GR7
                    :
        LD      GR1,GR7
                    :
        LD      GR4,GR7
                     :
CONST   DC      1234  ◀──── データを変更するときはここだけ変更すればよい。
```

**表7-2 LAD命令 と LD命令 の違い**

| 命令形式 | 語長(参考) | フラグレジスタ | 実行に必要な(予想)主記憶装置アクセス回数(*1) |
|---|---|---|---|
| LAD r,adr[,x] | 2 | 変更しない | 0 |
| LD r,ard[,x] | 2 | SF, ZFを変更する | 1 |

ところで、レジスタGR1に5を入れようとして次のようなプログラムを作成するとどうなるでしょうか。

```
LAD   GR1,=5
```

残念ながら、このプログラムではGR1には5はセットされません。GR1にはリテラル5が配置されたアドレスがセットされてしまいます。

(*1) 命令語のロードは除きます。

## ポイント

(1) 汎用レジスタに定数を設定する最も効率的な方法はLAD命令であるが、フラグレジスタを変更しないことに注意する。

(2) 同じ定数を何度も汎用レジスタに設定する場合には、LD命令の方がプログラミング上、都合が良いこともある。

## 演習問題7-1-1

(1) レジスタGR7に定数12345を設定するプログラムを、(a)LAD命令を使う、(b)ラベルとLD命令を使う、(c)リテラルとLD命令を使う、という方法で記述してください。

(2) すべての汎用レジスタをラベルVALUEの領域に格納された値と同じにするプログラムを2通りの方法で記述し、比較してください。

(3) 次の2つのプログラムが生成する機械語の違いについて説明してください(やや難：最適化機能はないものとします)。

```
     LD      GR1,TEN
     LD      GR2,TEN
     LD      GR3,TEN
                :
TEN  DC      10
```

プログラム1

```
LD      GR1,=10
LD      GR2,=10
LD      GR3,=10
```

プログラム2

(4) 次のプログラムにどのような最適化が考えられますか。最適化機能とは、プログラムの動作を変更しないで、そのプログラムの効率が上がるように機械語を生成する機能です。

```
LD      GR1,=10
LD      GR2,=10
LD      GR3,=10
```

### 7.1.2 ゼロクリア

汎用レジスタの値をゼロにしたいという処理はよく出てきます。この処理を**ゼロクリア**(zero clear)といいます。CASL IIでは、次のようなプログラムによってこれを実現できます。

まず、7.1.1項で学習した定数をセットする方法を用いると、次の方法が考えられます。これらの違いについては、もう説明できますね。

```
                LAD     GR1,0
                LD      GR1,ZERO
                LD      GR1,=0
                 ∫
ZERO            DC      0
```

しかし、ゼロをセットするには、上記の方法よりも効率の良い、次の方法が考えられます。これらの方法は、ゼロをセットするというよりも、「計算した結果をゼロにする」という方法であると考えられます。

```
SUBA    GR1,GR1
SUBL    GR1,GR1
XOR     GR1,GR1
```

最初の2つは、同じ数を引いてゼロを作り出しています。算術計算(符号付き)でも、論理計算(符号無し)でも同じ数を引けばゼロになります。XOR命令は、排他的論理和を取る命令でした。同じ数同士の排他的論理和はすべてのビットをゼロにします。

この3つの命令の中で、もっとも効率(実行速度)が良いと考えられるのは、XOR命令でしょう。減算の算術演算よりもビット演算の方が、計算手順が少ないため、効率がよくなると考えられます。

これらの違いを一覧にして表7-3に示します。

**表7-3　GR1をゼロクリアするさまざまな命令**

| 命　令 | | 語長 | 主記憶装置へのデータアクセス | 機械語の実行速度(予想) |
|---|---|---|---|---|
| LAD | GR1,0 | 2 | 0 | 中速 |
| LD | GR1,ZERO | 2 | 1 | 低速 |
| LD | GR1,=0 | 2 | 1 | 低速 |
| SUBA | GR1,GR1 | 1 | 0 | 高速 |
| SUBL | GR1,GR1 | 1 | 0 | 高速 |
| XOR | GR1,GR1 | 1 | 0 | かなり高速 |

## ポイント

汎用レジスタをゼロクリアするには、XOR命令が最も効率が良いが、他の方法もある。

## 演習問題7-1-2

(1) すべての汎用レジスタをゼロクリアする最も効率の良いと思われるプログラムを書いてください。

(2) GR3を、偶数のときだけゼロクリアして、奇数のときには1にする処理を1命令で書いてください。

(3) GR4が0か1のどちらかの値しか取らないという条件のときに、GR4が1のときにはGR4をゼロクリア、GR4が0のときにはGR4に1を設定する処理を1命令で書いてください(やや難)。

## 7.2 汎用レジスタを用いた演算

COMET IIでは、一つの機械語命令で表7-4に示す汎用レジスタ間演算ができます。ただし、r,r1,r2は汎用レジスタ(GR0〜GR7)、xは指標レジスタ(GR1〜GR7)、cは定数を意味します。

表7-4 汎用レジスタを用いた演算

| | 演算パターン | 使用する命令 | 語長 | フラグレジスタ |
|---|---|---|---|---|
| レジスタの値の複写 | r1 ← r2<br>r ← x | LD r1,r2<br>LAD r,0,x | 1<br>2 | OF=0,SF,ZFは設定される<br>変更無し |
| レジスタへの定数加減算 | r ← x ± c | LAD r,c,x | 2 | 変更無し |
| レジスタ同士の加減算<br>(r2に定数を入れて<br>おけば、定数加減算<br>にもなる) | r1 ← r1 ± r2 | ADDA r1,r2<br>ADDL r1,r2<br>SUBA r1,r2<br>SUBL r1,r2 | 1<br>1<br>1<br>1 | OF, SF, ZFの<br>すべての<br>フラグが<br>変更される |

### 7.2.1 汎用レジスタの値の複写

汎用レジスタの値を別の汎用レジスタに複写するには、LD命令が最も効率が良いでしょう。たとえば、GR3の内容をGR5に複写する命令は次のようになります。

```
LD      GR5,GR3
```

LD命令のオペランドには、すべての汎用レジスタを指定できますから、レジスタGR0の値をGR7に複写したい場合には、次のようにします。

```
LD      GR7,GR0
```

この他に、効率は悪くなりますが、汎用レジスタの値を別の汎用レジスタに複写するには、LAD命令を利用するという方法もあります(*1)。たとえば、GR3の内容をGR5に複写する命令は次のようになります。

```
LAD     GR5,0,GR3
```

「0,GR3」によって生成される実効アドレスは、「0＋GR3の値」、すなわち、GR3の値ですから、それがGR5に複写されます。しかし、実際には、LD命令を用いた方が効率がよいため、この命令を使うことはまずないでしょう。

さらに、効率を気にしないのなら、スタックを使うことによっても、汎用レジスタの値を別の汎用レジスタに複写できます。

```
PUSH    0,GR3
POP     GR5
```

つまり、PUSH命令によってスタック上に積み上げたGR3の値を、POP命令でレジスタGR5で受け取るのです。しかし、この方法は、2つの機械語命令を実行するだけでなく、主記憶装置に1度データを格納し、さらにそこから取り出すという方法を使っているので効率が悪く、現実的には第12章で学

(＊1) LD r1,r2は語長が1語ですみますが、LAD r,0,x は語長が2語必要になります。

習するサブルーチンのレジスタ待避とレジスタ交換を同時に行うこと以外では使用することはないでしょう。しかし、「このようなこともできるんだぞ」ということを知っておくことは、機械語命令の理解を深める上ではとても大切なので、ここでの内容は理解しておいてください。

ところで、LAD命令およびPUSH命令は、生成される実効アドレスをベースにするため、転送元のレジスタには指標レジスタ(GR1〜GR7)しか指定できません。つまり、この場合、転送元にはGR0は指定できませんので、注意が必要です。

## ポイント

汎用レジスタの値を他の汎用レジスタに複写する方法には、さまざまなものがあるが「LD r1,r2」が最も効率が良い。

## 演習問題7-2-1

(1) すべての汎用レジスタをGR0と同じ値にするプログラムを記述してください。

(2) 「LAD　GR5,0,GR3」によって、GR3の内容がGR5に複写される仕組みについて説明してください。また、「LD　GR5,0,GR3」はどのような意味になりますか。

(3) 次の2つのプログラムは主記憶装置上にあるNの値をレジスタGR0とGR1に複写するものです。それぞれの効率を比較してください。

```
LD   GR0,N
LD   GR1,N
```

プログラム1

```
LD   GR0,N
LD   GR1,GR0
```

プログラム2

### 7.2.2 汎用レジスタへの定数加算

汎用レジスタへ特定の定数を加える(たとえば、GR3に5を加える)には、LAD命令を使う方法と、ADDA命令・SUBA命令・ADDL命令・SUBL命令を使う方法の2つがあります。

LAD命令を用いて、汎用レジスタに特定の値を加えるには、次の形式を使います。

```
LAD     GR3,5,GR3     GR3 ← GR3 ＋ 5
LAD     GR4,－1,GR4   GR4 ← GR4 － 1
```

このときに注意しなければいけないのは、LAD命令がフラグレジスタを変更しないということです。たとえば、「LAD GR4,－1,GR4」はレジスタGR4の値をカウントダウンしますが、この命令はフラグレジスタの値を変更しないので、「GR4の値がゼロになったらどこかへ分岐する」というようなプログラムを記述することができないのです。

そこで、フラグレジスタを変更させるために算術演算命令を用いて、次のようにする方法もあります。

```
      ADDA GR3,FIVE
      SUBA GR4,MONE
           :
FIVE  DC   5
MONE  DC   －1
```

ラベルを使う方法

```
ADDA   GR3,=5
SUBA   GR4,=－1
```

リテラルを使う方法

この方法であれば、フラグレジスタを変化させるので、ループなどのチェックに使うことができます。しかし、上記の方法では、命令語が2語であるだけでなく、主記憶装置からデータを取ってくるというアクセスが発生します。

そこで、あらかじめ他の汎用レジスタに値を入れておき、汎用レジスタ同士の演算の機能を利用するという方法があります。たとえば、レジスタGR6には5、GR7には1をあらかじめセットしておきます。

```
LAD      GR6,5
LAD      GR7,1
```

この準備をしておけば、次の命令でGR3の値に5を加え、GR4の値から1を引くというプログラムになります。それぞれの機械語は1語しか占有しません。

```
ADDA     GR3,GR6     GR3 ← GR3 ＋ GR6 = GR3 ＋ 5
SUBA     GR4,GR7     GR4 ← GR4 － GR7 = GR4 － 1
```

第9章で学習するループ中のカウントアップやカウントダウンでは、この方法が特に効果的です。算術演算(ADDA, SUBA)か論理演算(ADDL, SUBL)は、必要に応じて使い分けると良いでしょう。一般に、通常のデータであれば算術演算、アドレスデータなどであれば論理演算を使います。

**表7-5　汎用レジスタへの定数加算**

| 命令形式 | 語長 | 主記憶装置アクセス | フラグレジスタ | 説　明 |
|---|---|---|---|---|
| LAD　r,定数,r | 2 | 0 | 変更しない | 効率は良いが、フラグレジスタを変更しない |
| ADDA　r1,r2 | 1 | 0 | 変更 | r2にあらかじめ定数をロードしておく必要がある |
| ADDA　r,ラベル | 2 | 1 | 変更 | ラベルに定数をセットしておく |
| ADDA　r,=リテラル | 2 | 1 | 変更 | 生成される機械語は「ADD r,ラベル」と同じ |

## ポイント

(1) 汎用レジスタr1に定数を加えるには「ADDA　r1,r2」命令(算術加算の場合)が効率が良いが、r2にあらかじめ定数を入れておく必要がある。しかし、ループの内部など何度も繰り返される場合には、この方法は抜群の効率を発揮する。

(2) 汎用レジスタに余裕がないときには、LAD命令が効率が良いが、フラグレジスタを変更しない。

## 演習問題7-2-2

(1) 次の命令をリテラルを使わないで1命令で書いて下さい。

(a) GR1←GR1－1 (b) GR2←GR2＋12 (c) GR3←GR3＋00AB(16) (d) GR4←GR4－02FD(16)

(2) 次の命令の誤りを指摘してください。

(a) LAD GR0,1,GR0

(b) LAD GR1, －1, GR1 　GR1から1を引いてそれが0になったらSKIPへ
　JZE SKIP

(3) レジスタGR1～GR7の値をすべて1つカウントダウンするプログラムをリテラルを使わないで7命令以内で書いてください。なお、フラグレジスタは特に変更しなくても結構です。

### 7.2.3 汎用レジスタへ定数加算(あるいは減算)をして、別の汎用レジスタに格納

汎用レジスタへ定数加算(あるいは減算)をして、別の汎用レジスタに格納する処理(r1←r2＋c)を1つの機械語で行う命令は、LAD命令しかありません。

```
LAD      GR2,5,GR3     GR2 ← GR3 ＋ 5
LAD      GR4,－1,GR6   GR4 ← GR6 － 1
```

このときに注意しなければいけないのは、LAD命令がフラグレジスタを変更しないということです。フラグレジスタの値を変更しないので、「GR4の値がゼロになったらどこかへ分岐する」というようなプログラムを記述することができません。また、指標レジスタにGR0は記述できませんので、r1←r2＋cにおいて、r2の部分にGR0は記述できません。

この処理を、これ以外の方法で記述しようとすると少しやっかいです。そこで、算術加算の場合を例として表7-6にまとめます(比較のため、LAD命令も含みます)。

**表7-6 汎用レジスタへ定数加算をして、別のレジスタに格納する方法( GR2←GR3＋5, 算術加算 )**

| プログラム<br>(方法) | 全体の<br>命令語長 | フラグ<br>レジスタ | 破壊される<br>レジスタ個数 | 説　明 |
|---|---|---|---|---|
| LAD GR2,5,GR3 | 2 | 変更しない | 0 | 単独で使うなら最も効率が良いがフラグレジスタが変化しない。 |
| LAD GR1,5<br>LD GR2,GR3<br>ADDA GR2,GR1 | 4 | 変更 | 1<br>(この例では<br>GR1) | 計算用にレジスタを1つ余分に使う。しかし、1度、(この例では)GR1に定数をセットしておき、あとで何度もその定数加算を行う場合には(たとえばループ)、最も効率がよい。(第9章参照)。 |
| LD GR2,GR3<br>ADDA GR2,FIVE<br>FIVE DC 5 | 3(*1) | 変更 | 0 | フラグレジスタを変更させる必要があり、ただ、一度だけこの計算を行う場合には、この方法が良い。 |
| LD GR2,GR3<br>ADDA GR2,=5 | 3(*1) | 変更 | 0 | |

表7-6の「破壊されるレジスタ個数」について説明します。「レジスタが破壊される」とは、その汎

(*1) 値5を保持する語は全体の命令語長には数えません。

用レジスタに以前の値と異なる別の値を設定することを言います。つまり、以前の値が使えなくなるために「以前の値が破壊された」という意味合いで使います。決して、汎用レジスタそのものを壊すわけではありません。

なお、表7-6の2つ目の例で、汎用レジスタGR1の値をその処理の前後で保存しておきたい場合には、PUSH命令でその値をスタックに待避させ、POP命令でスタックから戻します(7.3.2項参照)。

```
PUSH    0, GR1      レジスタ待避
LD      GR1, =5
LD      GR2, GR3
ADDA    GR2, GR1
POP     GR1         レジスタ復元
```

## ポイント

汎用レジスタに加減算をして別の汎用レジスタに加えるには、1命令ならLAD命令(フラグレジスタを変更しないことに注意)を用い、フラグレジスタを変更する必要があるなら、LD命令と算術命令を組み合わせる。

## 演習問題7-2-3

(1) 次の計算を行うプログラムを、表7-6のすべての方法で記述してください。

(a) GR3←GR2＋5　(b) GR4←GR1－5　(c) GR7←GR6－1

(2) 次の計算を行うプログラムを、表7-6の中から可能な方法を選んで記述してください。
また、表7-6の方法を適用できないものがあるとすればその理由は何ですか。

GR7←GR0+6

(3) ループ(第9章参照)を行う場合、表7-6の2番目の方法が3番目と4番目に比べて、なぜ都合が良いのですか。説明してください(やや難)。

(4) 次のLAD命令の中でGR3←GR2－1となっていないものはどれか。

(a) LAD　GR3,－1,GR2
(b) LAD　GR3,65535,GR2
(c) LAD　GR3,#FFFF,GR2
(d) LAD　GR3,#8000,GR2

### 7.2.4 汎用レジスタ同士の加減算

汎用レジスタ同士の加減算(r1←r1±r2)は、ADDA, SUBA, ADDL, SUBLで行うことができます。こ

れは、次のパターンで行うことができます。

```
命令      r1,r2
```

r1, r2には、汎用レジスタを指定できますから、GR0も指定できます。

```
ADDA    GR1,GR0    GR1←GR1+GR0
```

上記の例は、GR1にGR0の内容を加算した例です。

## ポイント

汎用レジスタ同士の加減算は「算術命令 r1,r2」で行う。

## 演習問題7-2-4

(1) GR2←GR2+GR0のプログラムを書いてください。

(2) GR7の値をGR0〜GR6の各汎用レジスタに加えるプログラムを書いてください。

(3) 「ADDA GR3,GR3」とすると、GR3の値が倍になります。しかし、もっと効率の良いと思われる方法もあります。それを紹介してください。

# 7.3 その他

## 7.3.1 汎用レジスタの値の交換

汎用レジスタの値を交換するには、他の汎用レジスタを1つ余分に使ってもよいのであれば、次の方法があります。次のプログラムは、レジスタGR1とGR2を交換するものですが、レジスタGR3を退避用に用いていますので、以前にレジスタGR3に格納されていた値は、なくなってしまいます(GR3の値が破壊される)。

```
        LD      GR3,GR1
        LD      GR1,GR2
        LD      GR2,GR3
```

上記の方法は、汎用レジスタを1つ余分に使いますが、もっとも効率の良い方法であるといえるでしょう。GR3の値を保持しておきたい場合には、スタックにGR3の値を待避します。

```
        PUSH    0,GR3       レジスタ待避
        LD      GR3,GR1
        LD      GR1,GR2
        LD      GR2,GR3
        POP     GR3         レジスタ復元
```

しかし、この方法では5つの機械語命令になってしまいます。そこで、次の方法は、主記憶装置を介す方法の一つですが、データをいったん、主記憶装置に格納するため、他の汎用レジスタをまったく利用しません。もっとも直感的で分かりやすい方法と言えるでしょう。

```
        ST      GR1,TEMP
        LD      GR1,GR2
        LD      GR2,TEMP
                    :
TEMP    DS      1
```

この方法は、3命令(ST, LD, LD)と待避するデータを格納する1語分の領域を確保する1命令(DS)が必要になります。

一方、次のように、スタックにデータを待避させることによっても、他の汎用レジスタの値を使うことなしに、汎用レジスタの値を交換することができます。

```
        PUSH    0,GR1
        PUSH    0,GR2
        POP     GR1         GR2の値をGR1に復元する
        POP     GR2         GR1の値をGR2に復元する
```

しかし、この方法では4命令必要になり、さらにレジスタGR1, GR2の両方の値を主記憶装置(スタック)に置くことになり、主記憶装置とのアクセスが増えてしまいます。この方法はあまり現実的ではないように思えますが、第12章で学習するサブルーチン実行時のレジスタ待避などで使われることも

あります。また、7.3.3項も参照してください。

## ポイント

汎用レジスタの交換は、これといった方法がない。さまざまな方法を使えるようにすべきである。

## 演習問題7-3-1

GR6とGR7を交換するプログラムを、次の条件に従って、記述してください。

(a) スタックを利用して、レジスタGR1を退避用に使う。

(b) 汎用レジスタを退避用に使わず、退避用の記憶領域を使う(スタックを利用しない)。

(c) 汎用レジスタを退避用に使わず、スタックにデータを直接PUSH, POPする。

### 7.3.2 レジスタ退避

次のプログラムの「計算A」を行うとレジスタGR4～GR6の値が破壊(つまり変更)されてしまうとします。この「計算A」の前に格納されていたレジスタGR4～GR6の値を、計算終了後に元の値(「計算A」の実行前の値)に戻したいとします。この手順を**レジスタ退避**と呼びます。汎用レジスタの値をいったん別の場所に退避させて、必要になったら元に戻すというわけです。

たとえば、次の方法でも処理は可能です。

```
        ST      GR4,GR4S  }
        ST      GR5,GR5S  }  レジスタの値を領域（主記憶装置）に対避
        ST      GR6,GR6S  }

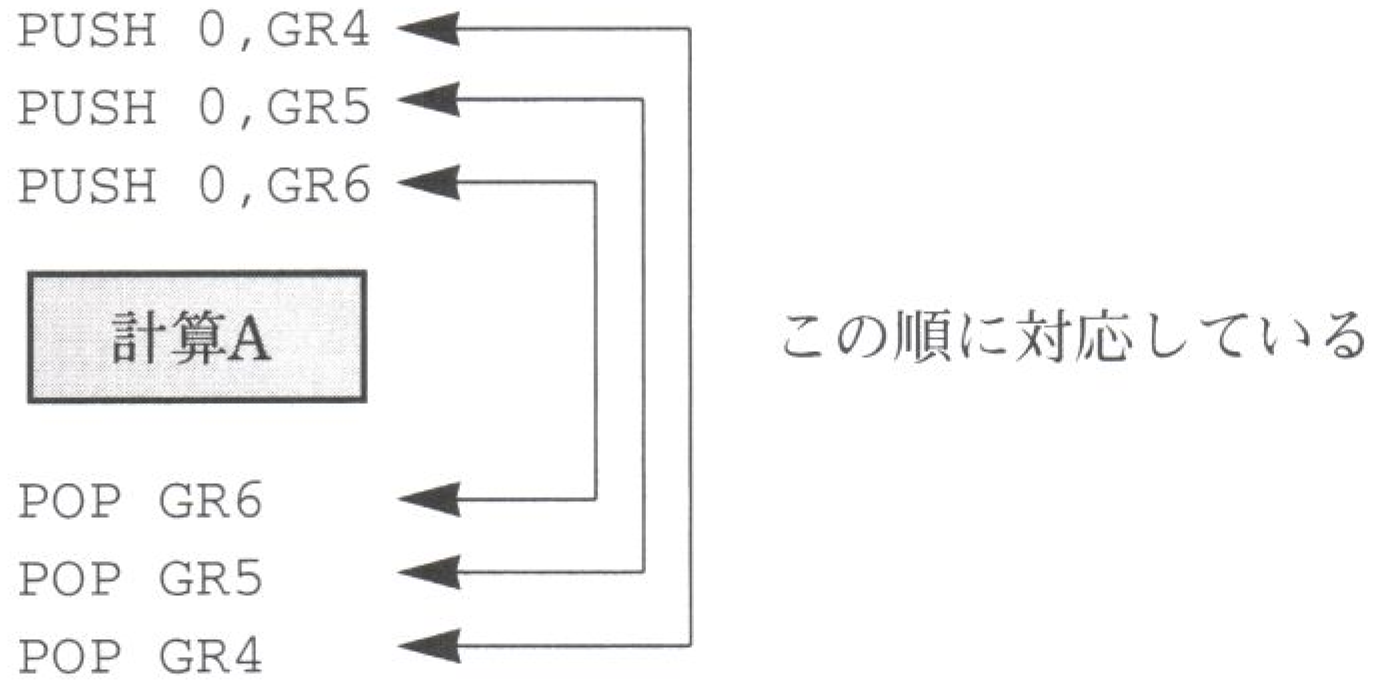
     [計算A(レジスタGR4～GR6の値が変更)]

        LD      GR4,GR4S  }
        LD      GR5,GR5S  }  主記憶装置に対避した値をレジスタに戻す
        LD      GR6,GR6S  }
                  ⋮
GR4S    DS      1
GR5S    DS      1
GR6S    DS      1
```

しかし、通常、このような方法は取らずに、スタックを使います[(*1)]。スタック上に、GR4～GR6の値を積み上げます。そして、「計算A」を行います。「計算A」終了後にスタックから値を戻します。

(*1) この主記憶に対避する方法と次に説明するスタックを用いる方法の間にはさほど違いがないように思えます。しかし、第12章で学習するサブルーチンを再帰的手続きで利用する場合には大きな違いがあります。詳細は、12.4節で説明します。

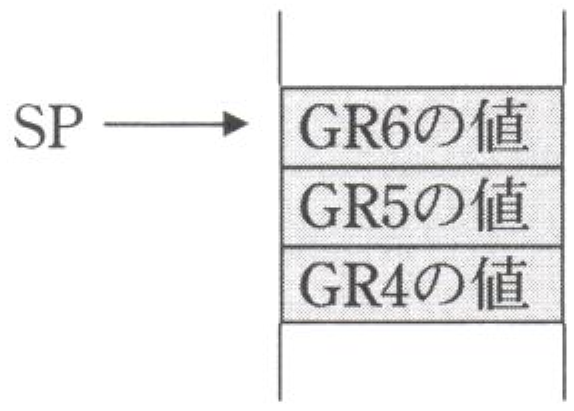
そのプログラムは次のようになります。「PUSH 0,GR4」は「0,GR4」の部分によって生成される実効アドレス(すなわGR4の値そのもの)をスタックにプッシュダウンします。




スタックに積んだ手順と戻す手順が逆になっていることに注意してください。なお、この方法では、PUSHのときに指標レジスタを指定するため、レジスタGR0の値をPUSHすることができませんので、注意してください(演習問題7-3-2(3)参照)。

なお、「計算A」の中でスタック操作を行ったら、せっかくスタックに積み上げたデータが破壊されると思う人もいるかも知れません。そこで、ここで行われる動作について説明しましょう。

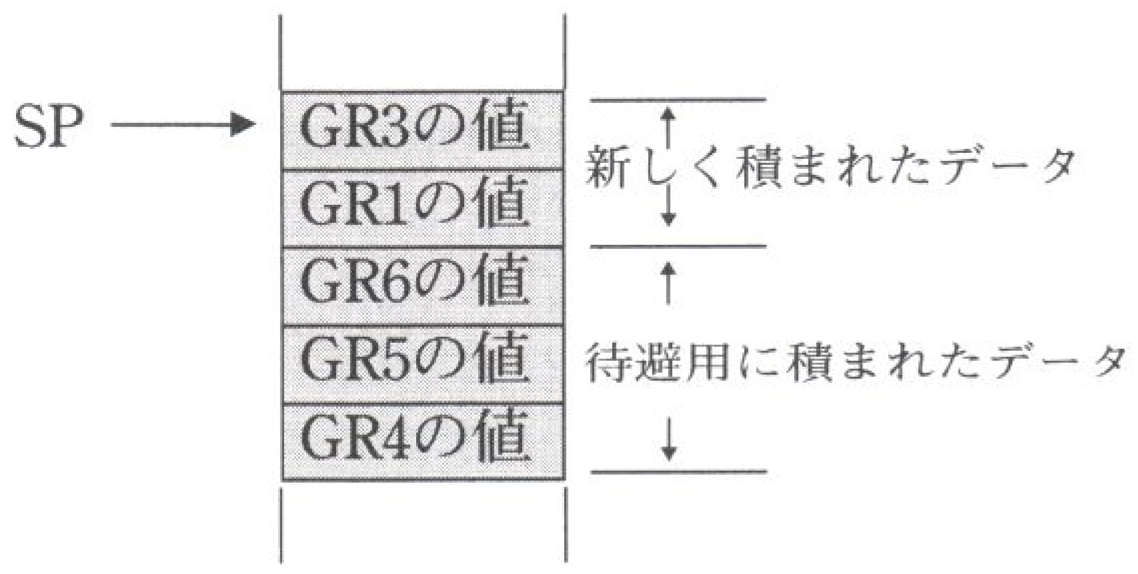
最初の3つのPUSHが行われた時点でスタックの内容は次のようになっています。




計算Aの中で次のようなスタック操作が行われたとします。

```
PUSH   0,GR1
PUSH   0,GR3
```

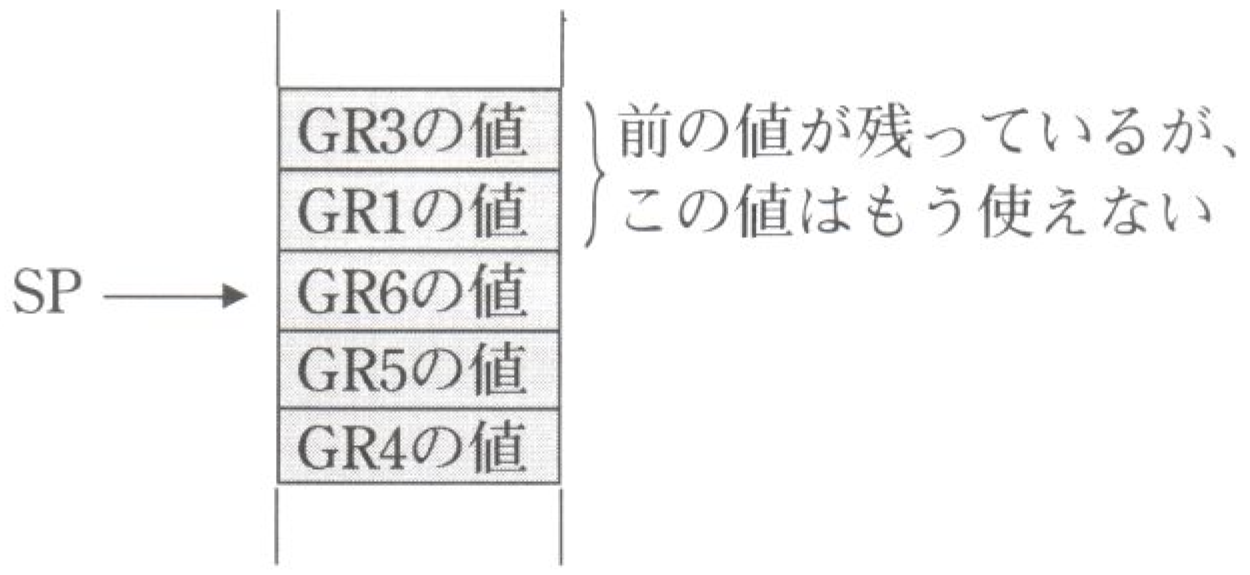
すると、スタックは次のようになります。




つまり、すでに積み上げられているデータの上に、GR1とGR3の内容が積まれるのです。この状態で、次の命令を実行します。

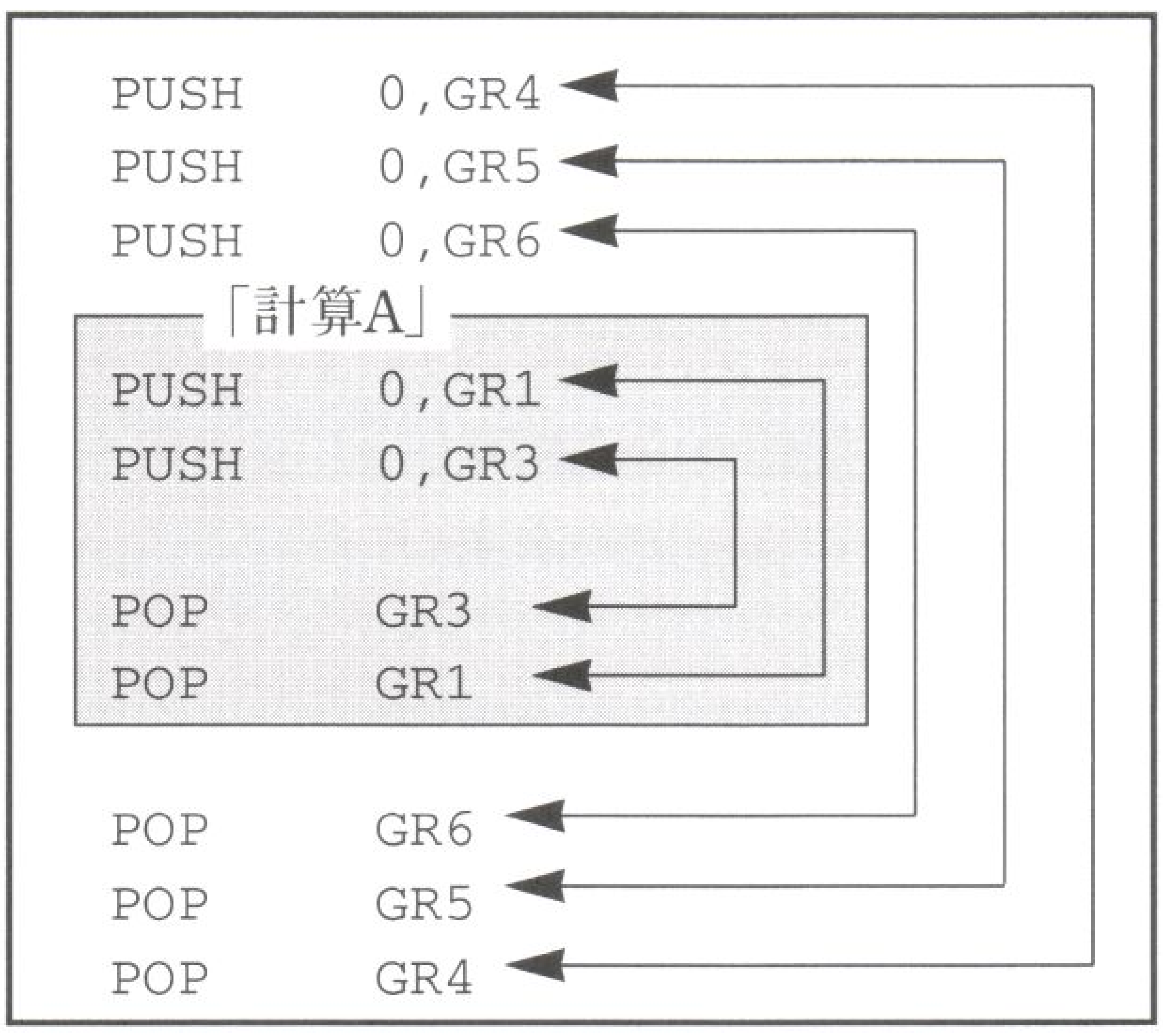
```
POP GR3
POP GR1
```

すると、スタックは次のようになります。

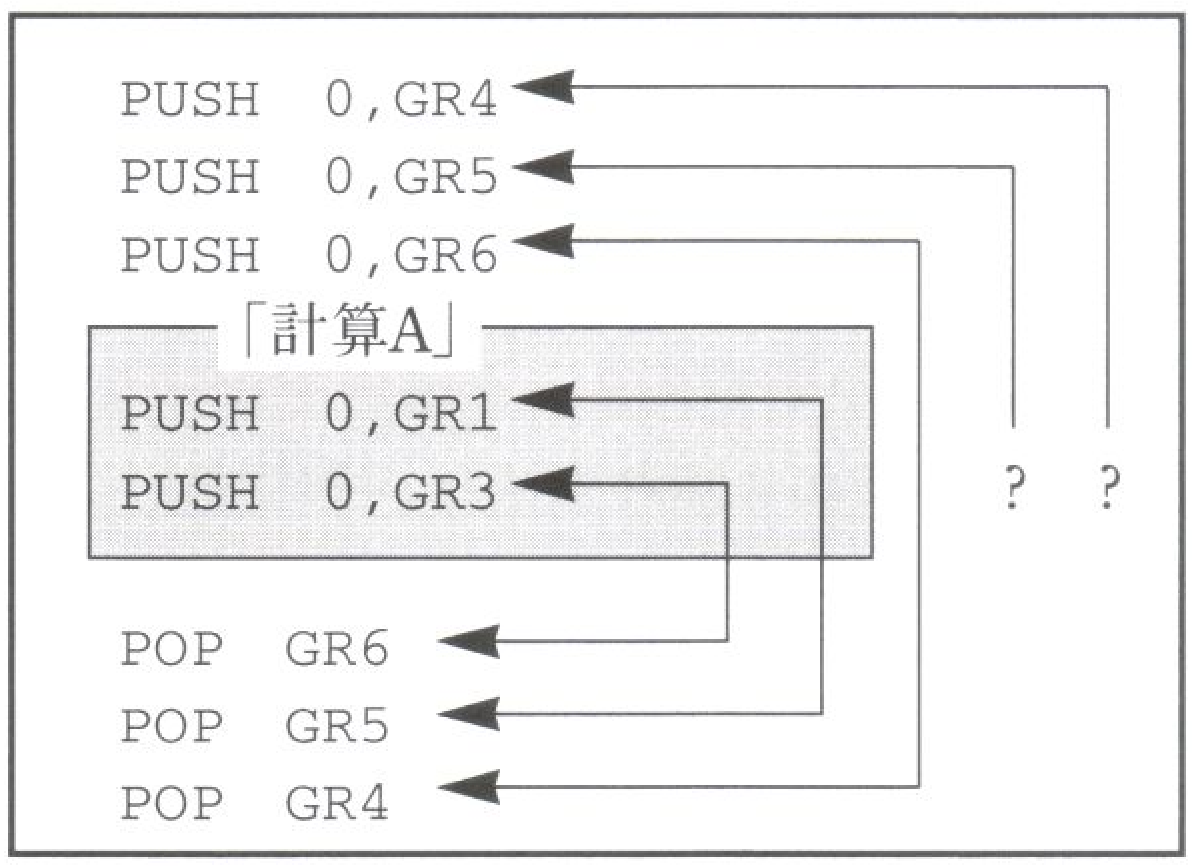



この状態で、POP操作をすれば、正しくデータが汎用レジスタに復元できます。

もし、「計算A」の中で、POP操作の記述をし忘れたとしましょう(右のプログラム)。すると、PUSHとPOPの対応関係が次のようになってしまいます。




正しいプログラム




対応が取れてないプログラム

この右のプログラムでは、うまくいきません。つまり、スタック操作は、その一連のプログラムの中で、PUSHとPOPの対応個数が一致していなければなりません。一致しないと、この例にあるようにプログラムのミスを誘発します。

## ポイント

汎用レジスタの待避はスタックを用いる。

PUSHとPOPの対応を正しく取る。

## 演習問題7-3-2

(1) レジスタGR1～GR7を待避させるプログラムを記述してください。

(2) レジスタGR0を待避させるプログラムを記述してください。他のレジスタの値を1つだけ破壊してもかまいません。

(3) 次のプログラムはレジスタGR0を他のレジスタを破壊することなしにプッシュダウンするプログラムです。空欄に命令を入れてください。

```
[ 1 ] GR1        GR1待避
LD    GR1,GR0
PUSH  [ 2 ]      GR0待避
何かの処理
POP   [ 3 ]      GR0復元
POP   GR1        GR1復元
```

### 7.3.3 汎用レジスタを退避し、同時に値を交換

汎用レジスタを退避し、同時に値を交換するには、次のようにすればよいでしょう。次の例は、GR4～GR6の値を退避し、復元時に、GR5とGR6の内容を交換している例です。スタックによる値の交換が意味を持つ例です。

```
PUSH    0,GR4
PUSH    0,GR5
PUSH    0,GR6
```

何らかの処理(レジスタGR4～GR6が破壊される)

```
POP GR5     GR6の値をGR5に格納
POP GR6     GR5の値をGR6に格納
POP GR4     GR4の値を復元
```

## ポイント

PUSH命令とPOP命令をうまく組み合わせると、レジスタ待避とレジスタの交換を同時に行うことができる。

## 演習問題7-3-3

「ある処理」後、GR1～GR3の値が「ある処理」前のGR4の値になっているようにするプログラムを記述してください(やや難)。

GR4待避

ある処理

GR1～GR3の値をGR4と同じにする

## 総合問題7

**[演習7-1]** C←20×A＋5×Bを行うプログラム(TE71.casl)を記述してください。

**[演習7-2]** 汎用レジスタ(GR0～GR7)の値をすべて1つ増やすプログラムをいくつかの方法で記述してその効率を比較してください。

**[演習7-3]** 汎用レジスタの値をすべて加えて、ラベルSUMの領域に格納するプログラムを作成してください(TE73.casl)。ただし、この処理の中ですべての汎用レジスタの値は保存されているものとします。

# 第8章 条件判断

条件判断は、CASL II理解の基礎になるだけでなく、情報処理技術者試験の問題を解く上でもキーとなる部分です。また、さまざまなテクニックを理解する上でも重要な概念です。本章では、8.1節で条件判断のプログラミングテクニックをまとめ、8.2節以降で実際のプログラミングについて解説していくことにします。

## 8.1 条件処理とフラグレジスタ

### 8.1.1 フラグレジスタとその役割

COMET IIでは、フラグレジスタを見て条件判断を行うことはすでに学習しました。そのフラグレジスタには、表8-1に示す3つがあり、それぞれ次の目的で使います。

表8-1 フラグレジスタの役割と用途

| 記号 | 名　前 | 意　味 | 用　途 |
|---|---|---|---|
| OF | Overflow Flag | 演算結果がオーバーフローしたことを示す | データのオーバーフローを検出、ビットのカウント、ループの制御(10,11章) |
| SF | Sign Flag | 演算結果が負(ビット番号15が1)になったことを示す | データの大小等のチェック、データの符号の判定、特定ビットの判定、奇遇判定、(主に10章) |
| ZF | Zero Flag | 演算結果が(全てのビットが0)ゼロになったことを示す | |

まず、SFとZFを利用して、2つのデータを比較するプログラムパターンを学習しましょう。OFの利用の仕方については、第6章(6.6.2頁)、8.2.3項、総合演習3.4、第10章、第11章を参考にしてください。

### 8.1.2 フラグレジスタによる条件分岐：CPA命令の例

さて、それでは、フラグレジスタの値によって処理を振り分けるプログラムの基礎について調べていきましょう。まず、レジスタGR1とGR2を比較するパターンについて調べてみましょう。まず、「CPA GR1,GR2」を実行して、フラグレジスタをセットします。このプラグレジスタをチェックして条件分岐するプログラムパターンを表8-2に示します。ところで、比較演算命令「CPA」と「CPL」では、OF(Overflow Flag)がセットされないことに注意してください。

**表8-2 比較命令(CPA GR1,GR2)の後の条件処理のプログラムパターン**

| 意味 | CPA GR1, GR2の後に指定する機械語 | 説明 |
|---|---|---|
| if GR1 = GR2 then jump to A | JZE A | ZF=1ならジャンプ |
| if GR1 ≠ GR2 then jump to A | JNZ A | ZF=0ならジャンプ |
| if GR1 > GR2 then jump to A | JPL A | SF=0かつZF=0ならジャンプ |
| if GR1 < GR2 then jump to A | JMI A | SF=1ならジャンプ |
| if GR1 ≧ GR2 then jump to A | JPL A<br>JZE A | SF=0かつZF=0ならジャンプ<br>ZF=1ならジャンプ |
| if GR1 ≦ GR2 then jump to A | JMI A<br>JZE A | SF=1ならジャンプ<br>ZF=1ならジャンプ |

たとえば、「if GR1 ＝ GR2 then jump to SAMESTEP」（もし、GR1とGR2の値が同じならラベルSAMESTEPへジャンプする）を実現するプログラムパターンは、次のようになります。

```
            CPA   GR1,GR2  フラグレジスタをセット
            JZE   SAMESTEP
                    ∫
                  GR1 ≠ GR2 のときの処理
                    ∫
SAMESTEP          GR1 = GR2 のときの処理
                    ∫
```

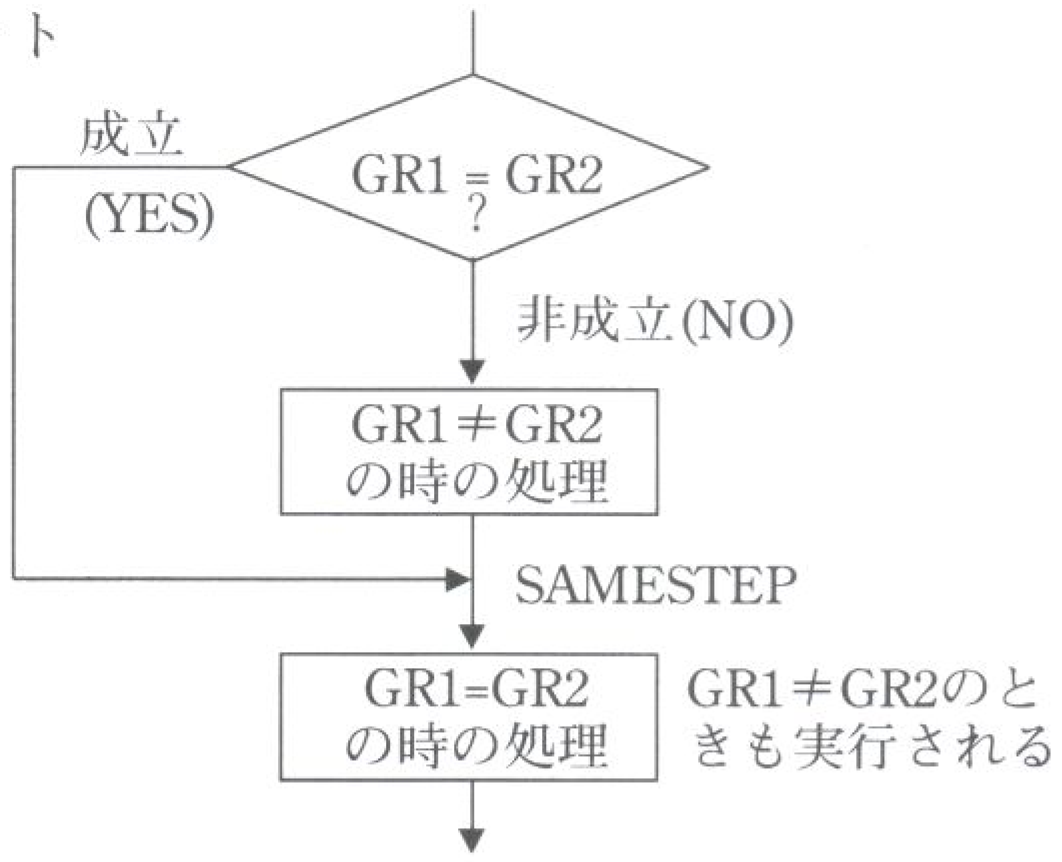



**図8-1 条件判断のフローチャート**

ただし、このプログラムパターンでは、「GR1＝GR2のときの処理」はGR1≠GR2のときも実行されるので注意してください。「GR1＝GR2のときの処理」と「GR1≠GR2のときの処理」の2つの処理を排他的に切り分けるには、次のプログラムパターンにします。

```
            CPA   GR1,GR2  フラグレジスタをセット
            JZE   SAMESTEP
                    ∫
                  GR1 ≠ GR2 のときの処理
                    ∫
            JUMP  EXITIF
SAMESTEP          GR1 = GR2 のときの処理
                    ∫
EXITIF      ここで合流
```

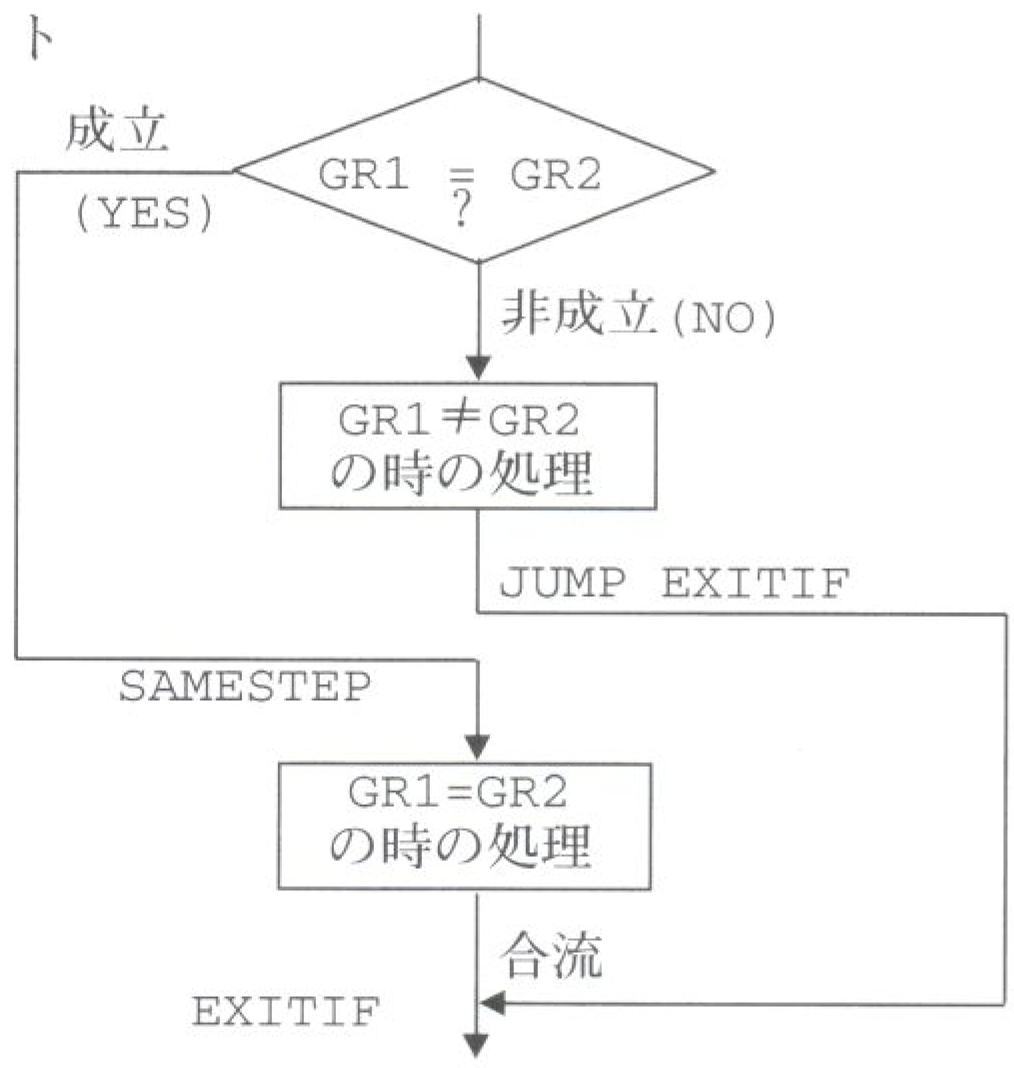



**図8-2 処理を振り分けた時のフローチャート**

図8-2のフローチャートでは、「GR1≠GR2」のときには「GR1≠GR2の時の処理」の終了後に「JUMP EXITIF」でジャンプしますので、「GR1=GR2の時の処理」は実行しません。

上記のプログラムを、次のようなプログラムとして作成する場合もあります。これは、「GR1=GR2の処理」を「本流」とは少し離れた別の場所に持っていった例です。そこから、また流れの本線にJUMP BACKPROCで戻ってきます。

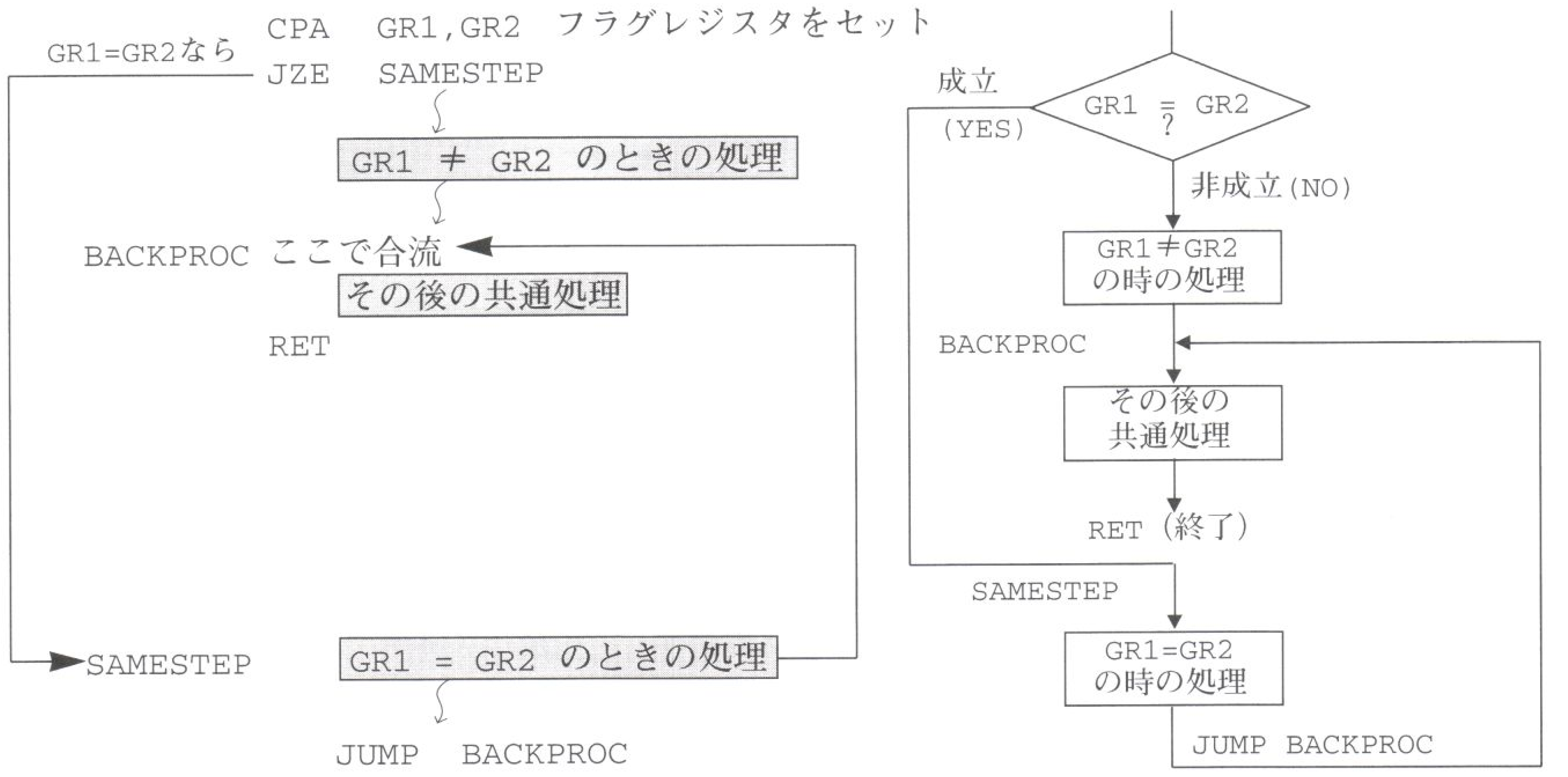



**図8-3 処理を振り分けた時のフローチャート(*1)**

ここで、大切なことは、この例が示すように、たかだか2種類の場合分けのプログラムであっても、実にさまざまな書き方があるということを理解するということです。

なお、図8-2、図8-3は、機械語の流れに沿ってフローチャートを書いているので、プログラム全体の流れが良く分かります。しかし、このようなアルゴリズムは、図8-4のように抽象化(*2)して書くことが多いので、各自必要な流れに展開できるようにしてください。

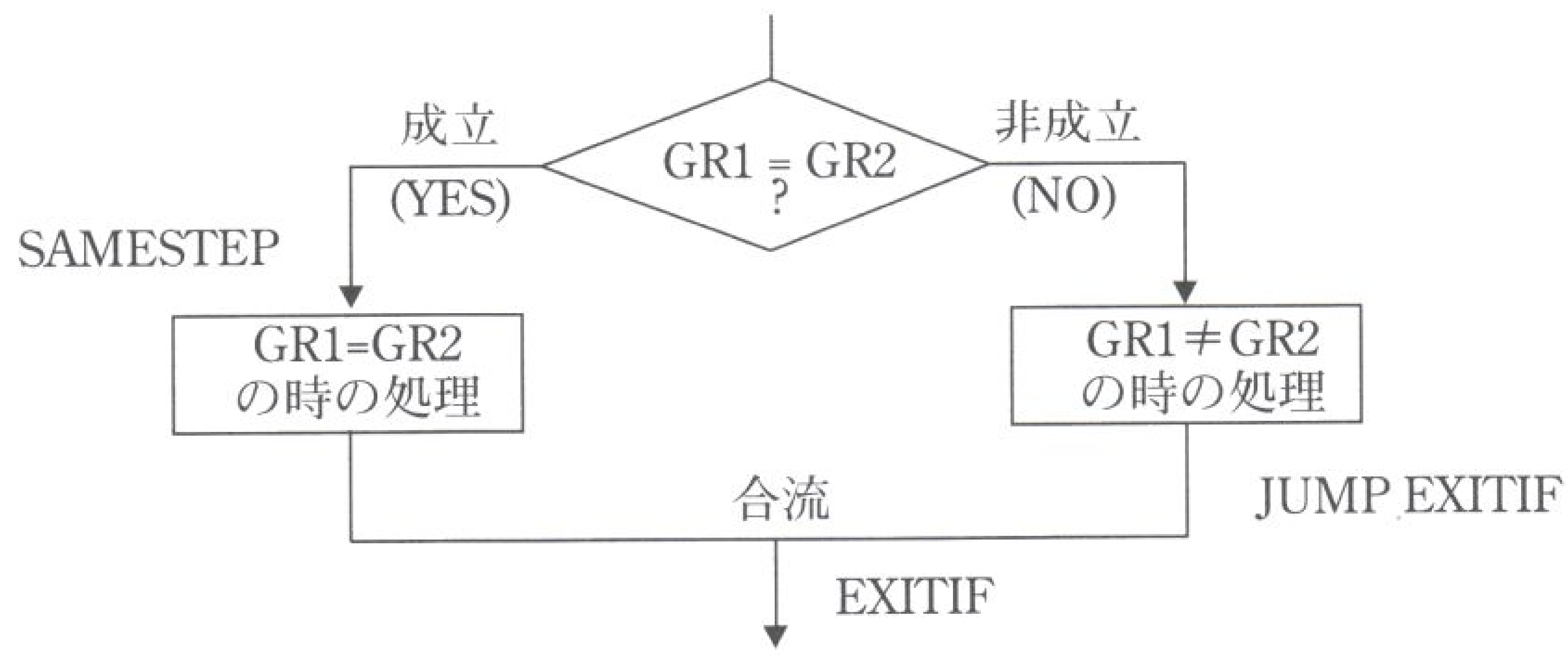



**図8-4 処理を振り分ける際の一般的なフローチャート**

(*1) 一般的にフローチャートは一番上が[開始](あるいは[START])で、一番下が[終了](あるいは[STOP])の形で記述すべきですが、この例では、アセンブラ言語の流れに沿って記述しているので、フローチャートの出口(RET)が途中にあります。

(*2) 要するに、ここでいう抽象化とは機械語の処理の直線的な流れを隠してしまうということです。図8-2、図8-3では、機械語の直線的な流れが連想できますが、図8-4では分かり難いですね。

さて、下の左のプログラムは、「もしGR1≧GR2ならPLUSZERO処理を、そうでないならMINUS処理をする」プログラムです。COMET IIでは、≧あるいは≦の処理では、この左の例のように「CPA GR1,GR2」の後に分岐命令を2つ使います。分岐命令自体は「フラグレジスタを変更しない」ため、「JZE PLUSZERO」の実行時に対しても、その2つ上の「CPA」命令でセットされたフラグレジスタが有効になります。

```
          CPA   GR1,GR2
          JPL   PLUSZERO
          JZE   PLUSZERO
          [MINUS処理]
          JUMP  SKIP
PLUSZERO  [PLUSZERO処理]
SKIP        ∫
```

もし、GR1≧GR2ならPLUSZEROへ

処理を反転させると → 単純になる

```
          CPA   GR1,GR2
          JMI   MINUS
                [PLUSZERO処理]
          JUMP  SKIP
MINUS           [MINUS処理]
SKIP              ∫
```

もし、GR1＜GR2ならMINUSへ

しかし、上の左のプログラムは分岐命令が2つあるので、あまりお勧めできるものではありません。考え方を逆にして、「もしGR1<GR2ならジャンプする」と考えて、上記の右のような分岐にすれば処理を単純にすることができます。もちろん、常にこんなに単純化できるとは限りませんが、アセンブラでプログラムを組む場合には、極力単純化したプログラムを作成すべきです。

ただし、次のように複数の分岐命令を利用するプログラムを組むこともあります。これは、GR1とGR2を比較し、「大・等・小」で処理を分けている例です。

```
          CPA     GR1,GR2
          JPL     BIGVAL     if GR1 > GR2 then jump to BIGVAL
          JZE     EQUVAL     if GR1 = GR2 then jump to EQUVAL
          [GR1 < GR2 のときの処理]
          JUMP    EXITPRS
BIGVAL    [GR1 > GR2 のときの処理]
          JUMP    EXITPRS
EQUVAL    [GR1 = GR2 のときの処理]
EXITPRS  命令       ; ここで合流
```

### 8.1.3 フラグレジスタによる条件分岐：ロード、算術演算、論理演算命令の例

フラグレジスタは、データのロード(LD命令)、算術演算命令(ADDA, SUBA)、論理演算命令(AND, OR, XOR)によってもセットされます。たとえば、これらの命令によって、データが汎用レジスタrにセットされたときは、次のプログラムパターンに従います。

**表8-3 ロード・算術演算・論理演算命令の後の条件処理のプログラムパターン**

| 意　　味 | ロード・演算命令の後に指定する条件分岐命令 | 説　明 |
|---|---|---|
| if r = 0 then jump to A | JZE A | ZF=1ならジャンプ |
| if r≠0 then jump to A | JNZ A | ZF=0ならジャンプ |
| if r > 0 then jump to A | JPL A | SF=0かつZF=0ならジャンプ |
| if r < 0 then jump to A | JMI A | SF=1ならジャンプ |
| if r≧0 then jump to A | JPL A<br>JZE A | SF=0かつZF=0ならジャンプ<br>ZF=1ならジャンプ |
| if r≦0 then jump to A | JMI A<br>JZE A | SF=1ならジャンプ<br>ZF=1ならジャンプ |

表8-3において、各条件分岐命令は、「LD, ADDA, SUBA, ADDL, SUBL, AND, OR, XOR」の各命令の直後においたときのものです。いくつか例を示します。

**表8-4 ロード、算術演算、論理演算と条件分岐命令の組み合わせの例**

| 命令の組み合わせ | LD GR1,N<br>JZE NISZERO | ADDA GR1,X<br>JPL PLUS | SUBL GR1,DATA<br>JMI BIG | AND GR1,=#0001<br>JZE EVEN |
|---|---|---|---|---|
| 説　明 | NをGR1にロードしそれがゼロならNISZEROへ | GR1にXを加えてその結果、正(>0)になったらPLUSへ | GR1からDATAを引いてSFが立ったら(32768以上)BIGへ | GR1の最下位ビットを抽出してゼロならEVENへ |

これらのプログラムは、「LD, ADDA, SUBA, ADDL, SUBL, AND, OR, XOR」の各命令が、フラグレジスタを変更することを利用しています。表8-3において、符号無しの値を扱うADDL, SUBLでは注意が必要です。これらの演算は、符号無しの数値を扱うので、演算結果は決して負になりません。しかし、フラグレジスタの定義を見ると、「演算結果の最上位ビットに1が立っている時にSFが1となる」と示されています。したがって、ADDL, SUBL演算を行った時のSFは次のようになります。

**表8-5 ADDL, SUBL演算を行った時のSFの値**

| | 演算結果の値 |
|---|---|
| SF=0となるとき | 0〜32767 |
| SF=1となるとき | 32768〜65535 |

表8-4の例から分かるように、ロード、算術演算、論理演算と条件分岐演算の組み合わせでは、「〜して分岐する」という処理を容易に組むことができます。

また、条件分岐命令ではフラグレジスタの値が変化しないことを利用して、次のように複数の条件分岐を記述することも可能です(8.1.2項でも紹介しました)。

```
          LD    GR1,DATA1
          JPL   PLUSPROC      DATA1が正のとき、PLUSPORCへジャンプ
          JZE   ZEROPROC      DATA1がゼロのとき、ZEROPORCへジャンプ
          [DATA1が負のときの処理]
          JUMP  ENDPROC
PLUSPROC  [DATA1が正のときの処理]
          JUMP  ENDPROC
ZEROPROC  [DATA1がゼロのときの処理]
ENDPROC
```

次の例は、第9章で学習するループを実現する際に使われるテクニックですが、レジスタGR3の値を一つカウントダウンして、正ならばLOOPCONTへジャンプするプログラムです。

```
          LAD   GR3,10
LOOPCONT  [ループの処理]
          SUBA  GR3,=1
          JPL   LOOPCONT
```

詳細は、第9章で学習しますが、このプログラムによって、ループの処理の部分が10回実行されます。

### 8.1.4 特定の汎用レジスタの値によって分岐させたい場合

汎用レジスタの値によって、処理を分岐させるにはどのようにしたら良いでしょうか。効率の良い方法は、LD命令によって自分を自分にロードさせ、フラグレジスタを変更させるのが良いでしょう。たとえば、GR6の値がゼロなら、ZEROGR6に分岐するには次のようにします。

```
LD    GR6,GR6
JZE   ZEROGR6
```

## ポイント

(1) 条件判断は、CPA, CPL, LD, ADDA, ADDL, SUBA, SUBL, AND, OR, XORでフラグレジスタをセットして分岐命令で分岐する。

(2) ≧、≦は1度に判別できないので、その反対(<,>)のロジックで判定できないか考える。

(3) 算術演算・論理演算命令と分岐命令をうまく組み合わせると「〜して分岐する」という処理が容易に組める。

## 演習問題8-1

次の各処理を行うプログラムがその下に示されています。空欄に適当な文字を入れてください。なお、「if 〜 then jump to ラベル」という指示は、「もし、〜の条件が成立すればラベルへジャンプし、成立しなければその命令の次の命令を実行せよ」ということです。

(a) if GR4≠GR5 then jump to LB1

```
CPA      GR4,[ 1 ]
[ 2 ]    LB1
```

(b) if (GR4=GR5) かつ (GR6=GR7) then jump to LB2

方法1

```
        CPA    GR4,GR5
        [ 1 ]  [ 1 ]
        JNZ    [ 1 ]
        CPA    GR6,GR7
        [ 2 ]  SKIP
SKIP    命令
         ⁝
LB2     命令
```

方法2

```
        CPA    GR4,GR5
        [ 3 ]  CONTI
NEXT    命令
         ⁝
CONTI   CPA    GR6,GR7
        [ 4 ]  LB2
        JUMP   NEXT
LB2     命令
```

(c) if (GR0=GR1) または (GR2=GR3) then jump to LB3

```
CPA    GR0,GR1
[ 1 ]  LB3
CPA    GR2,GR3
[ 2 ]  LB3
```

(d) if GR0>GR1 then jump to LB4
if GR0=GR1 then jump to LB5
if GR0<GR1 then jump to LB6

```
CPA    GR0,GR1
[ 1 ]  LB4
[ 2 ]  LB5
[ 3 ]  LB6
```

(e) if GR0>GR1 then jump to LB7
if GR0=GR1 then jump to LB8
if GR0<GR1 then jump to LB7

```
CPA    GR0,GR1
[ 1 ]  LB8
[ 2 ]  LB7
```

(f) if GR1が偶数 then jump to EVEN

```
AND    GR1,=[ 1 ]
[ 2 ]  EVEN
```

(g) if GR1が4の倍数 then jump to DOUBLE

```
AND      GR1,=[ 1 ]
[ 2 ]    DOUBLE
```

(h) if GR1が8の倍数 then jump to D8

```
AND      GR1,=[ 1 ]
[ 2 ]    D8
```

(i) GR1を1つカウントダウンしてゼロになったらEXITPROCへジャンプ

```
SUBA     GR1,=[ 1 ]
[ 1 ]    EXITPROC
```

(j) GR1を1つカウントアップして100になったらOKPROCへジャンプ

```
[ 1 ]    GR1,=1
[        2        ]
[ 3 ]    OKPROC
```

# 8.2 単純な条件判断プログラミング

本節では、単純なプログラムを例題にとり、CASL IIでの分岐命令のプログラミングを学習していきます。

## 8.2.1 絶対値

> **例題 8-1**
>
> ORGDATAに格納されている数値の絶対値を計算し、ANSWERに格納するプログラムを作りなさい。

データをレジスタGR1に格納し、その値が負ならば、符号を反転させればよいわけですね。このプログラムの流れは、次のようになるでしょう。

```
                       GR1 ← ORGDATA
   GR1≧0のとき         もし、GR1≧0なら、SKIPへジャンプ
  ┌────────────
  │                    GR1 ← GR1の符号を反転
  └──────────▶ SKIP   GR1 → ANSWER
                       RET
```

さて、負の数が格納された汎用レジスタの絶対値を求めるには、どのようにしたら良いのでしょうか。COMET IIでは、負の数を表現するのに、2の補数表現を用いています。2の補数表現を求めるには、基本的には、次のようにします(*1)。

**【2の補数の求め方：その1】**

> (1) データのビットの0と1をすべて反転する(XOR命令)
> (2) データに1加える(ADDA命令)

この処理をプログラムで行うには、次のようにします。データのビットの0と1を反転するには、全てのビットが1のデータ($FFFF_{(16)}$)と排他的論理和(XOR)を取ればよいのです(6.2.2項参照)。したがって、(1)、(2)の手順のプログラムは次のようになります。

```
XOR     GR1,=#FFFF
ADDA    GR1,=1
```

さて、これを素直にプログラム化したものが次のプログラムです。

(*1)「基本的には」と断ったのは、この例では、オーバーフロー時の対策をしていないからです。これについては、本項の最後のFAQで解説します。

絶対値を求めるプログラム(ABS.casl)

```
ABS       START
          LD      GR1,ORGDATA     GR1←ORGDATA
          JPL     SKIP            if GR1>0 then jump to SKIP
          JZE     SKIP            if GR1=0 then jump to SKIP
          XOR     GR1,=#FFFF      GR1のビットを反転させる
          ADDA    GR1,=1          GR1に1を加える
SKIP      ST      GR1,ANSWER      GR1→ANSWER
          RET                     ;プログラムの終了
ORGDATA   DC      −120
ANSWER    DS      1
          END
```

なお、このプログラム(ABS.casl)は本書に付属のCD-ROMの中にサンプルプログラムとして納められていますので、付属のCASLDVでぜひ動作を確認して下さい。

**【2の補数の求め方：その2】**

2の補数はその値を0から引いても求められます。

(1) その値を0から引く

たとえば、GR3に格納されている値の2の補数をGR2に求めるには次のようにします。

```
XOR       GR2,GR2    GR2←0
SUBA      GR2,GR3    GR2←GR2−GR3
```

この方法は汎用レジスタを2つ用いるうえに、減算という処理を行っています。これらの処理は「ビットを反転して1を加える」という処理に比べると効率が悪いと考えられます。

**例題 8-2**

さて、プログラム(ABS.casl)では、GR1≧0ならばSKIPへジャンプしているので、分岐命令(JPL命令とJZE命令)が2つ続いています。そこで、GR1が負ならば、GETABSラベルへジャンプするようにして、条件判断を1回だけ行うようにしてください。

プログラム(ABS1.casl)は次のようになります。

```
ABS1      START
          LD      GR1,ORGDATA     GR1←ORGDATA
          JMI     GETABS          if GR1<0 then jump to GETABS
          JUMP    SETDATA         (GR1≧0のときは)jump to SETDATA
GETABS    XOR     GR1,=#FFFF      GR1のビット反転
          ADDA    GR1,=1          GR1に1を加える
SETDATA   ST      GR1,ANSWER      GR1→ANSWER
          RET                     ;プログラムの終了
ORGDATA   DC      -120
ANSWER    DS      1
          END
```

**例題 8-3**

例題8-2のプログラム(ABS1.casl)では、分岐命令は1つ(JMI命令)で済みましたが、結局その後にJUMP命令を置いているので、全体の命令数は変わりません。例題8-1のプログラムの分岐命令を1つにする方法はありませんか。

例題8-1のプログラムは、データが0のときは符号反転をしなくてもよいので、JZE命令でそのチェックもしているのです。しかし、データが0のときは、符号反転して+1しても、オーバーフローした1が切り捨てられるのでその結果はゼロになります。

```
                   0000000000000000  ← もとのデータ
                          ↓
                   1111111111111111  ← ビット反転
                 +                1  ← +1する
                 ─────────────────
オーバーフローの  1 0000000000000000
分は切り捨てられ ↗       ↓
る                 0000000000000000  ← 結局、ゼロになる!
```

したがって、ゼロのときに、2の補数に変換するという処理を行っても結果は変わりません。そこで、次のように書くことができます(ABS2.casl)。

```
ABS2      START
          LD      GR1,ORGDATA
          JPL     SKIP          ORGDATA>ならSKIPへ
          XOR     GR1,=#FFFF  ┐ GR1の2の補数を求める(ORGDATA=0のときも
          ADDA    GR1,=1      ┘                     この処理は行われる)
SKIP      ST      GR1,ANSWER
          RET
ORGDATA   DC      -120
ANSWER    DS      1
          END
```

もちろん、このプログラムは、データがゼロの時にも符号反転してしまいますが、プログラムは多少短くなっています。

---

*FAQ* 汎用レジスタの値を+1するのにADDAを使っていますが、ADDLではいけませんか。ADDAとADDLの使い分けが、いまいち、よく分からないのですが。

*Answer*

確かに、COMET IIの場合、すべての計算が同じ16ビット長ですので、算術演算命令(ADDA, ADDL, SUBA, SUBL)の計算結果は見かけ上まったく同じになります(*1)。そのため、違いがあまり感じられません。このプログラムでは、ADDAの部分をADDLに変えても同じように動作します。それ

---

(*1) 異なる語長のデータを混在して扱うシステムでは違いがでてきます。C言語では、unsigned short(16ビット符号無しデータ)、signed short(16ビット符号付きデータ)、unsigned long(32ビット符号無しデータ)、signed long(32ビット符号付きデータ)があり、これらのデータの相互のデータ変換には特に注意が必要になります。

は、負の数の計算メカニズムとして2の補数を用いているからです。

しかし、プログラム(ABS.casl, ABS1.casl, ABS2.casl)で正しく動作しないデータが1つだけあります。それは、－32768です。試しに、プログラムABS.caslのORGDATAを－32768にして実行してみましょう。

ADDAでの実行結果：

```
PR    Instruction          SP   OSZ  GR0   GR1   GR2   GR3   GR4   GR5   GR6   GR7   Explanation
0000  LD       GR1,000D  7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  000D:[h:8000, s:-32768, u:32768]
0002  JPL      000A      7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  no jump
0004  JZE      000A      7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  no jump
0006  XOR      GR1,000F  7530  ---  8029  7FFF  98BE  E784  CAE1  BD6C  ACD6  F2AE  000F:[h:FFFF,u:65535]^=> GR1:[u:32767]
0008  ADDA     GR1,0010  7530  OS-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  0010:[h:0001, s:1]+=> GR1:[s:-32768]
000A  ST       GR1,000E  7530  OS-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  GR1:[h:8000,s:-32768, u:32768] => 000E
000C  RET                7530  OS-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  Return to OS
```

ちょっと分かりにくいのですが、最終的にレジスタGR1の値が符号付きで－32768になっていますね。－32768の絶対値を求めたのに、その答えが－32768になっています。しかし、これは、この方法ではやむを得ないのです。この理由は、符号付きデータの計算中にオーバーフローが起きてしまうからなのです。その証拠に、ADDAした時に、オーバーフローフラグ(OF)が立っていることがわかります。

このメカニズムを調べてみましょう。$-32768 = 8000_{(16)} = 1000000000000000_{(2)}$ですから、この2の補数(負の値)を求める処理である「ビット反転＋1」は次のようになります。

```
  1000000000000000  (=-32768(符号付き表現))
         ↓
  0111111111111111  ← ビット反転
+                1  ← +1
  ────────────────
  1000000000000000  ← 結局元へ戻る(=-32768(符号付き表現))
```

このときビット反転直後の値は$0111111111111111_{(2)} = 32767$です。これに1を加えると、$32767 + 1 = -32768$と負の値になります。これを符号付きデータとみなした場合、アンダーフローが起きているので、オーバーフローフラグが立ったのです。したがって、この現象が起きたことをオーバーフローフラグを調べることによってチェックすることができます。

一方、プログラムのADDAの部分をADDL(論理データ、すなわち符号無しデータの加算)に変えて実行してみましょう。

ADDLでの実行結果：

```
PR    Instruction          SP   OSZ  GR0   GR1   GR2   GR3   GR4   GR5   GR6   GR7   Explanation
0000  LD       GR1,000D  7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  000D:[h:8000,s:-32768, u:32768]
0002  JPL      000A      7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  no jump
0004  JZE      000A      7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  no jump
0006  XOR      GR1,000F  7530  ---  8029  7FFF  98BE  E784  CAE1  BD6C  ACD6  F2AE  000F:[h:FFFF,u:65535]^=> GR1:[u:32767]
0008  ADDL     GR1,0010  7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  0010:[h:0001, u:1]-=> GR1:[u:32768]
000A  ST       GR1,000E  7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  GR1:[h:8000,s:-32768, u:32768] => 000E
000C  RET                7530  -S-  8029  8000  98BE  E784  CAE1  BD6C  ACD6  F2AE  Return to OS
```

この場合には、32767＋1＝32768の計算が行われるだけで、オーバーフローフラグが立ちません。このように、多くの場合、ADDAとADDLの違いは、オーバーフローフラグの扱いです。なお、この場合でも、SF(サインフラグ)は立っています。絶対値を取っているのですから、数学的には負になったという印であるSFが立つはずがありません。しかし、COMET IIのSFはデータの最上位ビット(ビット番号15)が1であれば論理演算の場合でもSF＝1となるのです。そこで、このことを利用すれば、ADDL命令でもオーバーフローを起こしていることをチェックすることができます。

なお、算術演算命令では、符号付きのデータも符号無しのデータも見かけ上違いはないのですが、2倍などの目的のためにシフト命令などを使った場合には、SLLとSLAでは結果が完全に異なりますので注意が必要です。演習問題 11-5-2を参考にしてください。

## 演習問題8-2-1

(1) プログラムABS1を次のように変更したプログラムABS3(ABS3.casl)を作成して下さい。

```
ABS3     START
         LD    GR1,ORGDATA
         JMI   GETABS
SVPROC   ST    GR1,ANSWER
         RET
GETABS
          ∫
```

(2) 次の計算をしてNに格納するプログラムを記述してください。

(a) N ← −｜A｜
(b) P ← −｜A+B−C｜

### 8.2.2 点数の合計と判定

**例題 8-4**

国語、英語、数学の点数がそれぞれ、次の領域にセットされています。

```
KOKUGO    DC   80
EIGO      DC   54
SUUGAKU   DC   55
```

この3つの点数の合計が、180点以上なら合格とし、その点数をKEKKA(結果)に格納するプログラム SAITEN (SAITEN.casl)を作りなさい。なお、合計が180点未満の場合には、KEKKAには0を格納してください。

この問題は、例題8-5, 8-6のためのウォーミングアップの問題です。以下のプログラムでは、レジスタGR1に3科目の点数を合計し、その点数とGOUKAKUに格納されている点数(180)を比較し、合計点(GR1)の値が合格点に満たないならばNOTPASSへジャンプし、そうでないなら(つまり、合格したら9行目の)DTSETへジャンプし、そこでGR1(合計点)の値をKEKKAへ格納しています。6行目のJMI命令で8行目のNOTPASSへジャンプした時は、GR1の値をゼロにして9行目のST命令で0をKEKKAに格納します。

なお、このプログラムはあまり効率のよいものではありません。効率の悪さは、以降の例題8-5で改善していきます。

行番号　プログラム

```
 1 SAITEN    START
 2           LD      GR1,KOKUGO     GR1←国語の点数
 3           ADDA    GR1,EIGO       GR1←GR1+英語
 4           ADDA    GR1,SUUGAKU    GR1←GR1+数学
 5           CPA     GR1,GOUKAKU    合格点と比較
 6           JMI     NOTPASS        合格点に満たないのならNOTPASSへ
 7           JUMP    DTSET          結果格納へジャンプ
 8 NOTPASS   XOR     GR1,GR1        GR1←0
 9 DTSET     ST      GR1,KEKKA      GR1→結果
10           RET                   ; 終了
11 KOKUGO    DC      59             国語の点数
12 EIGO      DC      65             英語の点数
13 SUUGAKU   DC      89             数学の点数
14 GOUKAKU   DC      180            合格最低点
15 KEKKA     DS      1              結果(合格時：素点、不合格時：0点)
16           END
```

言うまでもないことですが、上記のプログラムの行番号は説明のためのもので、実際のプログラムには付いていません。

**例題8-5**

例題8-4では、JMI命令とJUMP命令が続いているなど、多少、効率の点で問題がありそうです。プログラム(SAITEN.casl)を見直して、もう少しすっきりしたプログラムにしてください。

例題8-4のプログラムでは、合格点(180)よりも低ければ「不合格」としていました。これを次のように考えればよいでしょう。つまり、不合格最高点(179点)よりも大きければ「合格」ですから、SKIPへジャンプして、GR1の値をKEKKAに格納します(8行目)。もし、「合格」でなければそのまますぐ下のプログラム(7行目)でGR1の値をゼロにします。

```
行番号 プログラム
  1 SAITEN1    START
  2            LD     GR1,KOKUGO     GR1←国語の点数
  3            ADDA   GR1,EIGO       GR1←GR1+英語
  4            ADDA   GR1,SUUGAKU    GR1←GR1+数学
  5            CPA    GR1,FGOUKAKU   不合格点と比較
  6            JPL    SKIP           合格点の場合にはSKIPへ
  7            XOR    GR1,GR1        不合格なのでGR1←0
  8 SKIP       ST     GR1,KEKKA      GR1→結果
  9            RET                 ; 終了
 10 KOKUGO     DC      59            国語の点数
 11 EIGO       DC      60            英語の点数
 12 SUUGAKU    DC      60            数学の点数
 13 FGOUKAKU   DC     179            不合格最高点
 14 KEKKA      DS       1            結果(合格時：素点、不合格時：0点)
 15            END
```

このようにちょっとロジックを変えると、条件判断命令のプログラムは、すっきりした形になります。実際の情報処理技術者試験などでは、さまざまなパターンで条件判断命令が出題されますので、いろいろなパターンに精通しておくことが大切です。

**例題8-6**

POINTに格納されているテストの点数を、次の点数で分類し、HANTEIに以下に示す文字コードをセットしなさい。

| テストの点数 | 判定 |
|---|---|
| 80 ～100 | A |
| 70 ～ 79 | B |
| 60 ～ 69 | C |
| 0 ～ 59 | X |

プログラム(PHANTEI.casl)を以下に示します。レジスタGR1に得点、レジスタGR2に判定結果を格納しています。流れは、プログラムのコメントを見ていただければ分かると思います。

```
PHANTEI START
        LD  GR1,POINT    GR1←POINT(得点)
        LD  GR2,='X'     GR2:それまでの判定を入れるレジスタ
        CPA GR1,=60      60点と比較
        JMI FOUND        60点未満ならFOUNDへ(この時点でGR2には'X'がセットされている)
        LD  GR2,='C'     判定を'C'にする
        CPA GR1,=70      70点と比較
        JMI FOUND        70点未満ならFOUNDへ
        LD  GR2,='B'     判定を'B'にする
        CPA GR1,=80      80点と比較
        JMI FOUND        80点未満ならFOUNDへ
        LD  GR2,='A'     判定を'A'にする
FOUND   ST  GR2,HANTEI     判定結果の格納
        RET            ; プログラムの終了
POINT   DC  75           点数
HANTEI  DS  1            結果保存場所
        END
```

## 演習問題8-2-2

(1) レジスタGR1を60点と比較しようとして「CPA GR1,60」としました。間違いを指摘してください。また、これはどのような命令になりますか。

(2) 「LD　GR2,='A'」で、実際にレジスタGR2に入る値を10進数と16進数で示してください。

(3) レジスタGR2に'A'をロードしようとして「LAD GR2,'A'」と記述して、アセンブラにかけたら、何が起こりますか(やや難)。

### 8.2.3 データのオーバーフロー

**例題 8-7**

2つの符号付きデータA, Bの和C←A＋Bを計算するプログラム(ADDOVA.casl)を作りなさい。ただし、データが大きいので、COMET IIの算術データの範囲を超えてしまうかもしれません。そのときには、オーバーフローなので、Cにはゼロを入れ、かつ、OVFに1をセットしなさい。なお、オーバーフローが発生していない時は、OVFには0をセットしなさい。

符号付きデータですから、算術加算ADDAを使います。オーバーフローしたかどうかは、算術加算後、分岐命令のJOV命令を用います。トレース結果と比較するため、アセンブラリストを示します。

```
 Machine       Source List (ADDOVA.casl)
Adr. Code     Line    Program
   |     |   1|ADDOVA    START
  0|1010 |   2|          LD   GR1,A       GR1←A
  1|0011 |
  2|2010 |   3|          ADDA GR1,B       GR1←GR1＋A
  3|0012 |
  4|6600 |   4|          JOV  OVERFLOW    もし、オーバーフローが発生していたらOVERFLOWへ
  5|0009 |
  6|3622 |   5|          XOR  GR2,GR2     GR2←0(ゼロクリア)
  7|6400 |   6|          JUMP FINAL       FINALへジャンプ
  8|000C |
  9|3611 |   7|OVERFLOW  XOR  GR1,GR1     GR1←0(ゼロクリア)
  A|1220 |   8|          LAD  GR2,1       GR2←1
  B|0001 |
  C|1110 |   9|FINAL     ST   GR1,C       GR1→C(Cの格納)
  D|0013 |
  E|1120 |  10|          ST   GR2,OVF     GR2→OVF(OVFの格納)
  F|0014 |
 10|8100 |  11|          RET               ; プログラムの終了
 11|4E20 |  12|A         DC   20000
 12|4E20 |  13|B         DC   20000
 13|0000 |  14|C         DS   1
 14|0000 |  15|OVF       DS   1
   |     |  16|          END
```

20000＋20000を算術加算で行うとオーバーフローが発生します。次のトレース結果(0番地からロード)でも、2番地のADDA命令を実行するとOFが立っていることが分かります。その結果、JOV命令でジャンプし、Cには0が、OVFには1が格納されています。

| PR | Instruction | | SP | OSZ | GR0 | GR1 | GR2 | GR3 | GR4 | GR5 | GR6 | GR7 | Explanation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | LD | GR1,0011 | 7530 | --- | 8029 | 4E20 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 0011:[h:4E20, s:20000, u:20000] |
| 0002 | ADDA | GR1,0012 | 7530 | OS- | 8029 | 9C40 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 0012:[h:4E20,s:20000]+=> GR1:[s:-25536] |
| 0004 | JOV | 0009 | 7530 | OS- | 8029 | 9C40 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | jump to 0009 |
| 0009 | XOR | GR1,GR1 | 7530 | --Z | 8029 | 0000 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | GR1:[u:0]^=> GR1:[u:0] |
| 000A | LAD | GR2,0001 | 7530 | --Z | 8029 | 0000 | 0001 | E784 | CAE1 | BD6C | ACD6 | F2AE | [h:0001, s:1, u:1] |
| 000C | ST | GR1,0013 | 7530 | --Z | 8029 | 0000 | 0001 | E784 | CAE1 | BD6C | ACD6 | F2AE | GR1:[h:0000,s:0, u:0] => 0013 |
| 000E | ST | GR2,0014 | 7530 | --Z | 8029 | 0000 | 0001 | E784 | CAE1 | BD6C | ACD6 | F2AE | GR2:[h:0001,s:1, u:1] => 0014 |
| 0010 | RET | | 7530 | --Z | 8029 | 0000 | 0001 | E784 | CAE1 | BD6C | ACD6 | F2AE | Return to OS |

次のトレース結果は、10000＋20000に対するものです。この場合には、オーバーフローは発生しないので、JOV命令でジャンプしていません。

| PR | Instruction | | SP | OSZ | GR0 | GR1 | GR2 | GR3 | GR4 | GR5 | GR6 | GR7 | Explanation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | LD | GR1,0011 | 7530 | --- | 8029 | 2710 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 0011:[h:2710, s:10000, u:10000] |
| 0002 | ADDA | GR1,0012 | 7530 | --- | 8029 | 7530 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | 0012:[h:4E20,s:20000] +=> GR1:[s:30000] |
| 0004 | JOV | 0009 | 7530 | --- | 8029 | 7530 | 98BE | E784 | CAE1 | BD6C | ACD6 | F2AE | no jump |
| 0006 | XOR | GR2,GR2 | 7530 | --Z | 8029 | 7530 | 0000 | E784 | CAE1 | BD6C | ACD6 | F2AE | GR2:[u:0]^=> GR2:[u:0] |
| 0007 | JUMP | 000C | 7530 | --Z | 8029 | 7530 | 0000 | E784 | CAE1 | BD6C | ACD6 | F2AE | jump to 000C |
| 000C | ST | GR1,0013 | 7530 | --Z | 8029 | 7530 | 0000 | E784 | CAE1 | BD6C | ACD6 | F2AE | GR1:[h:7530, s:30000, u:30000] => 0013 |
| 000E | ST | GR2,0014 | 7530 | --Z | 8029 | 7530 | 0000 | E784 | CAE1 | BD6C | ACD6 | F2AE | GR2:[h:0000,s:0, u:0] => 0014 |
| 0010 | RET | | 7530 | --Z | 8029 | 7530 | 0000 | E784 | CAE1 | BD6C | ACD6 | F2AE | Return to OS |

# 総合問題8

**[演習8-1]** MOJIに格納されている文字の種別を判定し、メッセージを表示するプログラム(TE81.casl)を作ってください。

| 10進数 | 16進数 | | メッセージ |
|---|---|---|---|
| 0 ～ 31 | 0 ～ 1F | 制御文字 | 'Control Key' |
| 32 | 20 | 間隔文字 | 'Space' |
| 48 ～ 57 | 30 ～ 39 | 数字 | 'Number' |
| 65 ～ 90 | 41 ～ 5A | 英大文字 | 'Upper' |
| 97 ～ 122 | 61 ～ 7A | 英小文字 | 'Lower' |
| 上記以外 | | 特殊文字 | 'Separator' |

**[演習8-2]** MOJIに格納されている文字が、大文字ならば小文字に変換するプログラムを作ってください(TE82.casl)。なお、大文字を小文字に変換するには$0020_{(16)}$とXORを取ります（表6-9、表10-1参照）。

**[演習8-3]** 2つのアドレスデータA, Bの和C←A+Bを計算するプログラム(TE83.casl)を作りなさい。ただし、データが大きいときには、COMET IIの論理データの範囲を超えてしまうかもしれない。そのときは、オーバーフローなので、Cにはゼロを入れ、かつ、OVFに1をセットしなさい。なお、オーバーフローが発生していない時は、OVFには0をセットしなさい。

# 第9章 繰り返し型のプログラミング

いよいよループのプログラミングを学習します。これにより、本格的なプログラムが組めるようになります。しかし、ループは、タイミングの考え方が必要になるため、初学者にとって難しい分野でもあります。本章では、簡単な例をふんだんに使って説明しているので、ぜひ理解してください。

## 9.1 ループの基礎パターン

**ループ**(loop)とは、第6章で学習したように、分岐命令で以前に実行した命令に戻ることによって同じ処理を繰り返すものです。この機能によって、プログラムの重複部分を大幅に減らすことができます。しかし、同じ処理を繰り返してばかりいたのでは、プログラムが終了しなくなってしまいます。このようなループを**無限ループ**(endless loop)と呼びますが、プログラムにはあってはならないことです。そこで、通常は条件分岐命令(*1)を用いて、ループから脱出します。

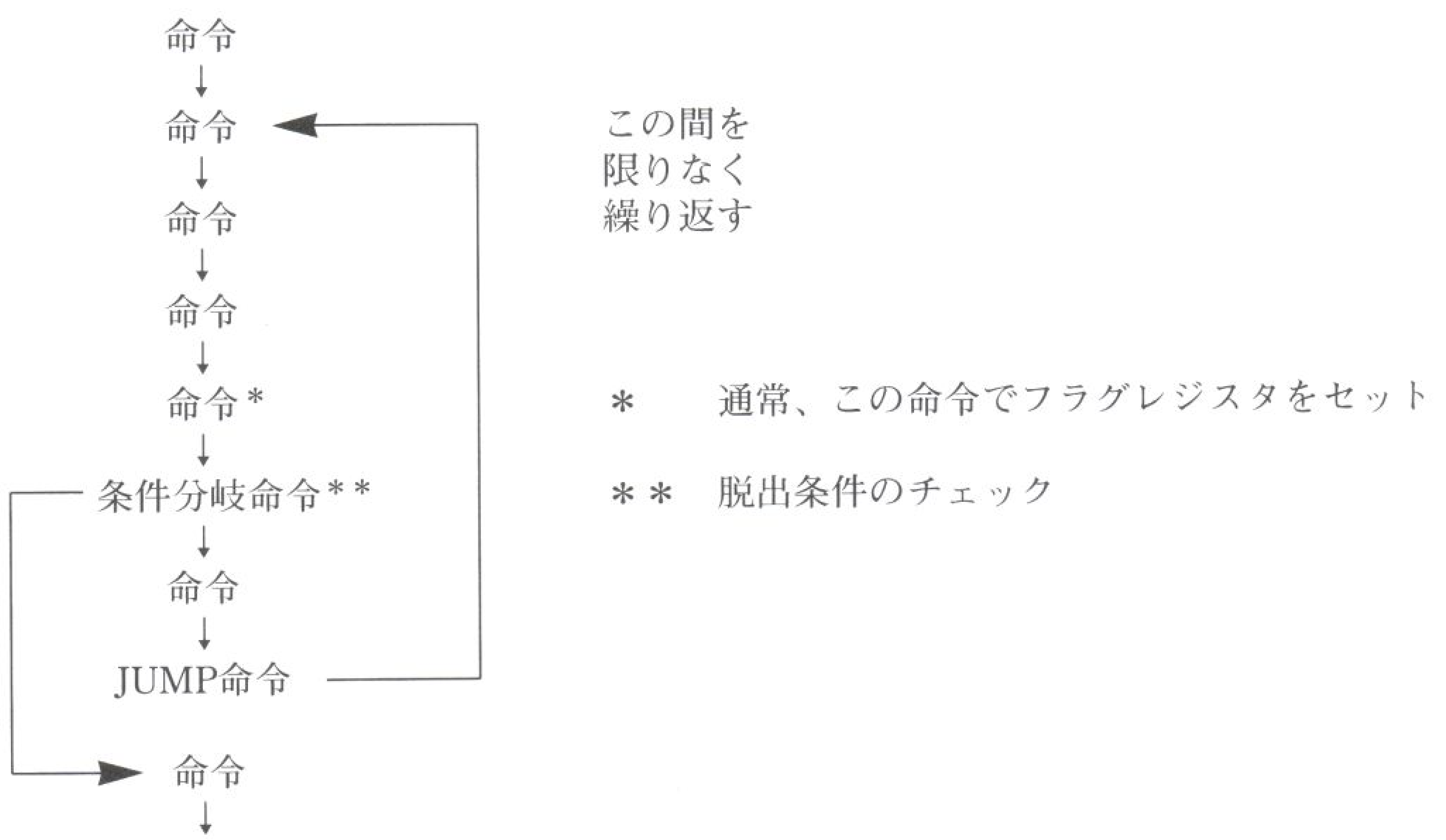



**図9-1 ループの概念**

図9-1の例では、途中の条件分岐命令によって、このループを脱出しています。ループを脱出する条件を「ループの脱出条件」と呼びますが、脱出条件の位置によっていくつかのループのパターンを形成することができます。脱出条件の例として、「GR1の値がゼロになったらループを終了」、「OF(Overflow Flag)が立ったら終了」などさまざまな脱出条件が考えられます。

(*1) 条件分岐命令については表6-17を参照して下さい。

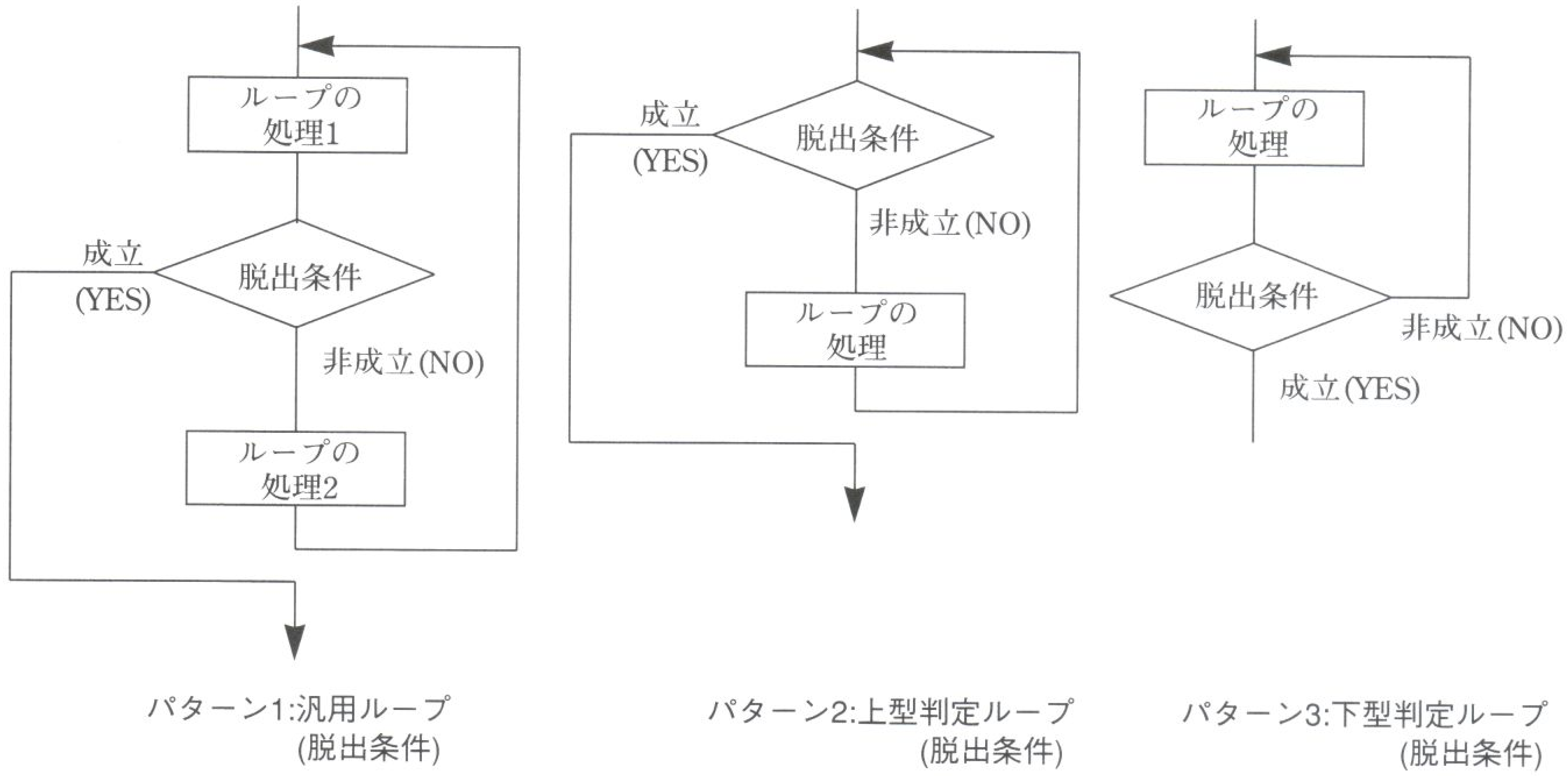



**図9-2　脱出条件によるループ**

この「脱出条件」の真偽を逆に考えると、「継続条件」と捉えることができます。継続条件の例として、「GR1の値が正の間ループを繰り返す」、「SF(Sign Flag)が立っている間ループを繰り返す」などさまざまな継続条件が考えられます。

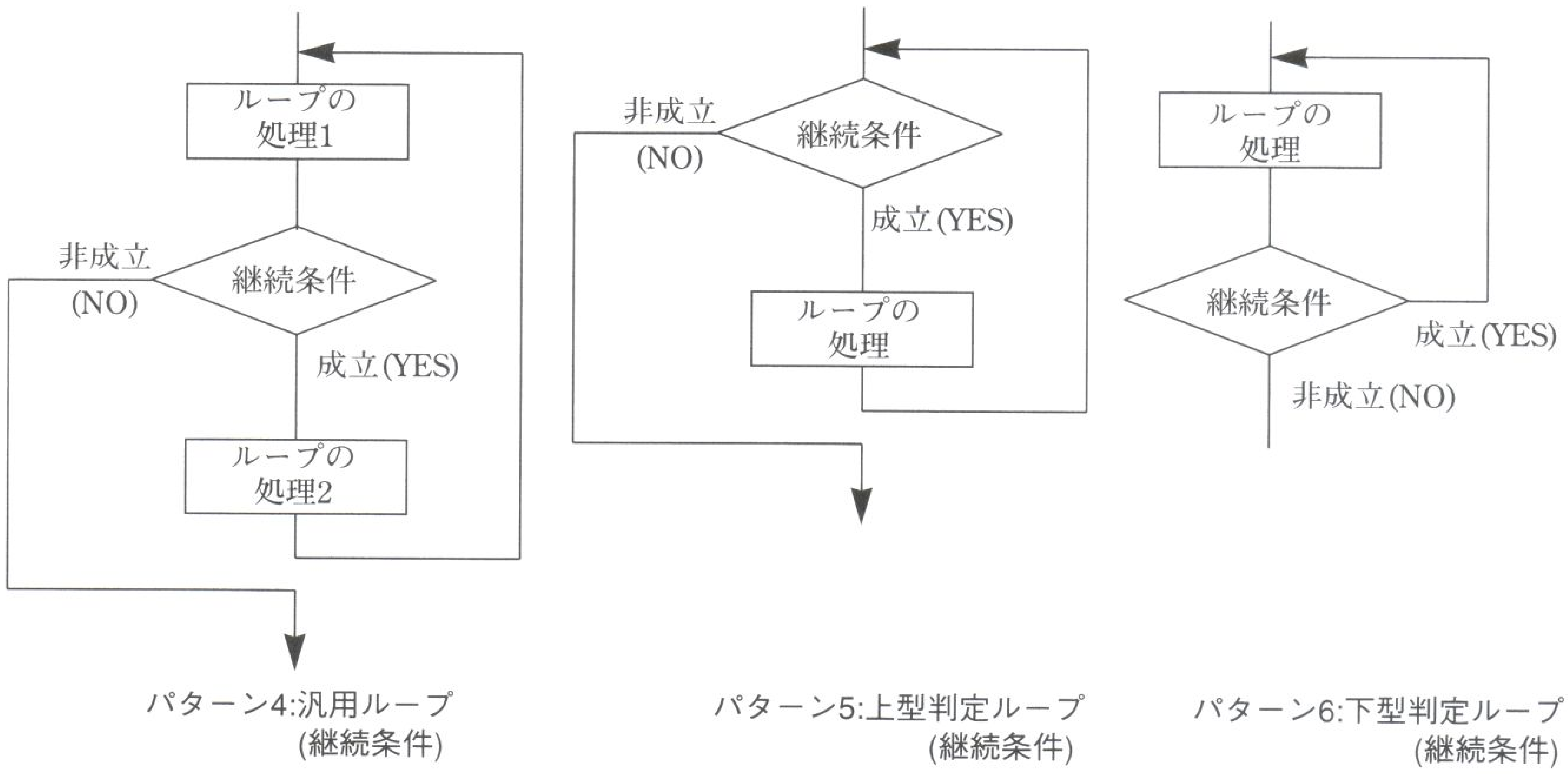



**図9-3　継続条件によるループ**

上記のように、さまざまなパターンのループが考えられます。実際の情報処理技術者試験などの問題を解く際には、このようなさまざまなループのパターンに慣れておく必要があります。

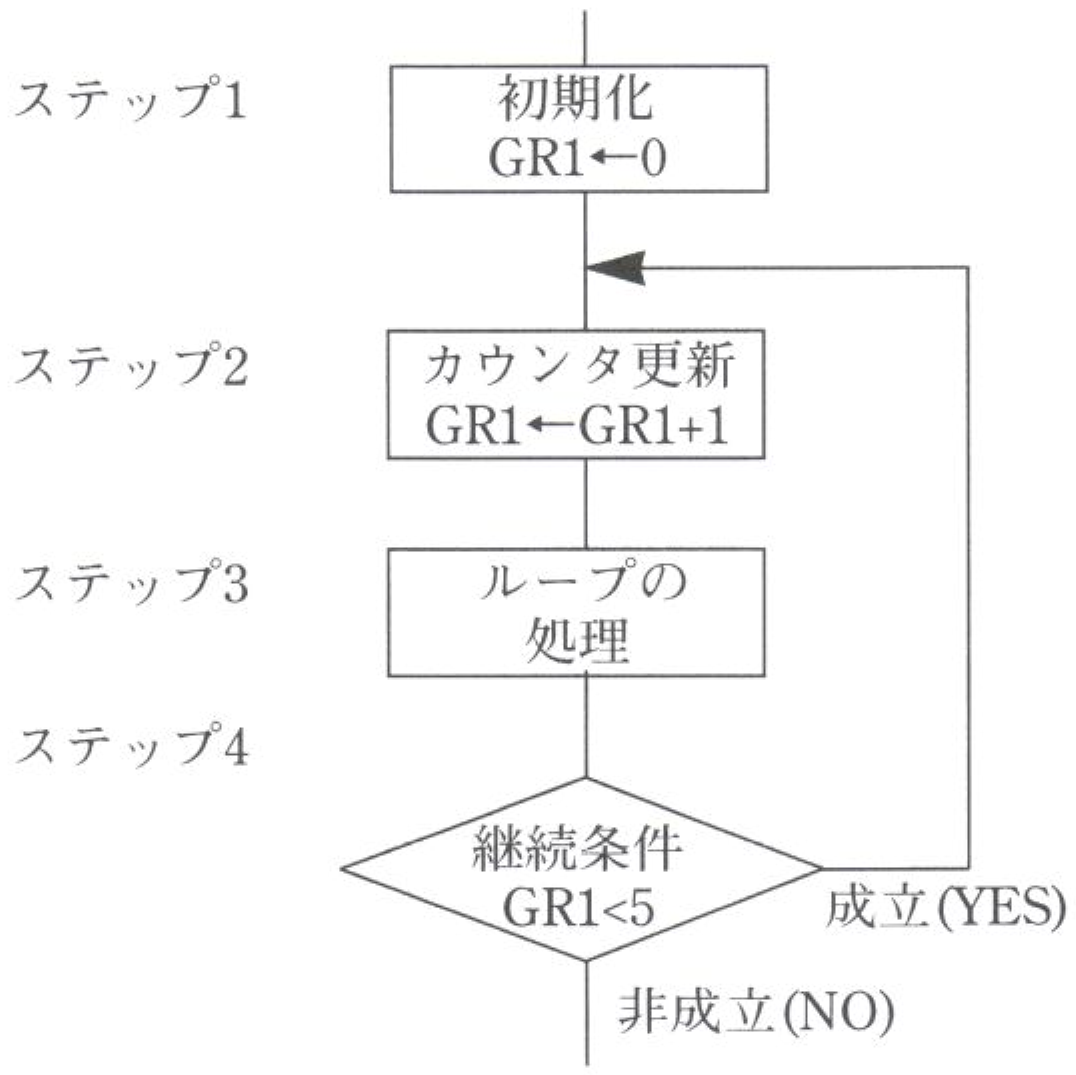
# 9.2 カウンタの更新によるループの基礎

## 9.2.1 GR1を1,2,3,4,5まで動かす：継続条件処理

汎用レジスタGR1の値をループの中で1,2,3,4,5と動かすプログラムについて考えてみましょう。一般に、ループのタイミングを取る値(この場合では、1,2,3,4,5)を保持する領域や汎用レジスタを**カウンタ**(counter)と呼びます。この場合、GR1がカウンタになります。

(1) 方法1：GR1を1,2,3,4,5まで動かす

このプログラムの流れは次のフローチャートのようになります。このフローチャートは基礎パターンとして、いつでも書けるようにしておいてください。これは、図9-3の「パターン6：下型判定ループ(継続条件)」を変形したものです。このフローチャート中の各処理に対して説明のため、ステップ1〜ステップ4という名前を付けます。




判定の様子

| GR1の値 | 継続条件 | 判 定 |
|---|---|---|
| 1 | 1＜5 | YES |
| 2 | 2＜5 | YES |
| 3 | 3＜5 | YES |
| 4 | 4＜5 | YES |
| 5 | 5＜5 | NO |

**図9-4 アルゴリズム(9.1)：GR1を1,2,3,4,5と動かすループ**

このときの動作の流れを調べてみましょう。まず、最初にステップ1でGR1に0がセットされます。最初に値をセットすることを**初期化**(initialization)と呼びます。この0の値は、次ページの表9-1のGR1の値が0の部分に相当します(網かけ部分)。GR1の値は、ステップ2の「カウンタ更新」の部分で＋1されて1になります。これを**カウンタの更新**と呼びます。その後、ステップ3の「ループの処理」に入りますから、その時のGR1の値(つまり、ループの最初のGR1の値)は1になります。次にステップ4の「継続条件」の条件判断が行われます。ここでは継続条件GR1＜5をチェックするので、比較演算「CPA GR1,＝5」の結果をOSZとして表9-1に示しています。GR1の値が1のときの結果を −S− と網かけで示しています。つまり、このとき、「CPA GR1,＝5」のフラグレジスタが、OF=0, SF=1, ZF=0となるので、これを-S-と表現しています。

継続条件が成立しループが続行するので、実行は再びカウンタ更新に移ります。ここで、GR1の値が再び1増加して(これをカウントアップと呼びます)、今度は2になります。

このように次々にGR1がカウントアップしてやがて、GR1の値は5になります。そのときの比較命令「CPA　GR1,＝5」の結果は、GR1＝5ですから--Zになります(網かけ)。

**表9-1 アルゴリズム(9.1)のループの動作の様子**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ステップ1 | 1 | | | | | | | | | | | ←ステップ実行の様子 |
| ステップ2 | | 2 | | 2 | | 2 | | 2 | | 2 | | ステップが実行されたこと |
| ステップ3 | | 3 | | 3 | | 3 | | 3 | | 3 | | をそのステップ番号で示す |
| ステップ4 | | | 4 | | 4 | | 4 | | 4 | | 4 | |
| GR1 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | |
| OSZ | ? | ? | -S- | | -S- | | -S- | | -S- | | --Z | CPA GR1,=5の結果 |

――― ループの継続 ―――▶ └▶ ループの終了

さて、ここで、表9-1のフラグレジスタの推移を見てみると、GR1の値が1〜4のとき-S-で、5のときに--Zに変わっています。タイミング的に見ると、フラグレジスタが-S-のときにループを継続するようにすばよいのです。

このロジックは、次の命令で実現できます(*1)。

```
        GR1←0
LOOP    GR1←GR1+1(カウンタ更新命令)
        ループの処理
        CPA   GR1,=5 GR1と5を比較する
        JMI   LOOP
```

ループ基本アルゴリズム1
(GR1=1,2,3,4,5)

それでは、実際のプログラム(LOOP1.casl)でこの様子を調べてみましょう。上記の基本アルゴリズムをプログラムとして組むと次のようになります。なお、このプログラムはカウンタGR1を更新するだけのもので、ループの処理の部分には、まだ、何の処理も想定していません。そこで、とりあえずNOP命令を入れておきます(*2)。

```
LOOP1   START
        XOR     GR1,GR1     GR1←0, GR1をカウンタリセット
LOOP    ADDA    GR1,=1      GR1(カウンタ)をカウントアップ
        NOP                 ;ループの処理
        CPA     GR1,=5      GR1:GR4(=5)比較
        JMI     LOOP        GR1 < 5のときLOOPへ
        RET
        END
```

このプログラムのアドレスを明記するため、プログラムリストも示します。

(*1) わかりやすくするためにアルゴリズムの紹介では一部日本語や数式を用いています。
(*2) NOP命令は実行しても「何もしない」という命令です。しかし、当然のことながら、プログラムを実行するとこのNOPの部分もトレースされます(本文のトレース結果参照)。NOPだけのトレースを見ると、ループの中で各レジスタの値がどのように推移しているかが良く分かります(この説明も本文に示されています)。

```
  Machine      Source List (LOOP1.casl)
Adr. Code Line Program
              1 LOOP1     START
  0  3611     2           XOR   GR1,GR1     GR1←0, GR1をカウンタリセット
  1  2010     3 LOOP      ADDA  GR1,=1      GR1(カウンタ)をカウントアップ
  2  0009
  3  0000     4           NOP               ; ループの処理
  4  4010     5           CPA   GR1,=5      GR1:GR4(=5)比較
  5  000A
  6  6100     6           JMI   LOOP        GR1 < COUNTのときLOOPへ
  7  0001
  8  8100     7           RET
  9  0001       (リテラルによって生成)
  A  0005       (リテラルによって生成)
              8           END
```

さて、このプログラムを0番地からロードしてトレースしてみましょう。NOP命令の部分に注目してください。また、SF=0となってループが終了していることを確認してください。

```
PR   Instruction      SP   OSZ  GR0  GR1  Explanation
0000 XOR   GR1,GR1    7530 --Z  8029 0000 GR1:[u:0]^=> GR1:[u:0]
0001 ADDA  GR1,0009   7530 ---  8029 0001 0009:[h:0001, s:1]+=> GR1:[s:1]
0003 NOP              7530 ---  8029 0001
0004 CPA   GR1,000A   7530 -S-  8029 0001 GR1:[s:1]? 000A:[h:0005, s:5]
0006 JMI   0001       7530 -S-  8029 0001 jump to 0001
0001 ADDA  GR1,0009   7530 ---  8029 0002 0009:[h:0001, s:1]+=> GR1:[s:2]
0003 NOP              7530 ---  8029 0002
0004 CPA   GR1,000A   7530 -S-  8029 0002 GR1:[s:2]? 000A:[h:0005, s:5]
0006 JMI   0001       7530 -S-  8029 0002 jump to 0001
0001 ADDA  GR1,0009   7530 ---  8029 0003 0009:[h:0001, s:1]+=> GR1:[s:3]
0003 NOP              7530 ---  8029 0003
0004 CPA   GR1,000A   7530 -S-  8029 0003 GR1:[s:3]? 000A:[h:0005, s:5]
0006 JMI   0001       7530 -S-  8029 0003 jump to 0001
0001 ADDA  GR1,0009   7530 ---  8029 0004 0009:[h:0001, s:1] +=> GR1:[s:4]
0003 NOP              7530 ---  8029 0004
0004 CPA   GR1,000A   7530 -S-  8029 0004 GR1:[s:4]? 000A:[h:0005, s:5]
0006 JMI   0001       7530 -S-  8029 0004 jump to 0001
0001 ADDA  GR1,0009   7530 ---  8029 0005 0009:[h:0001, s:1]+=> GR1:[s:5]
0003 NOP              7530 ---  8029 0005
0004 CPA   GR1,000A   7530 --Z  8029 0005 GR1:[s:5]? 000A:[h:0005, s:5]
0006 JMI   0001       7530 --Z  8029 0005 no jump
0008 RET              7530 --Z  8029 0005 Return to OS
```

NOP命令の部分だけを抜き取ると次のようになっています。

```
PR   Instruction      SP   OSZ GR0  GR1  GR2  GR3  GR4  GR5  GR6  GR7  Explanation
0003 NOP              7530 --- 8029 0001 98BE E784 CAE1 BD6C ACD6 F2AE
0003 NOP              7530 --- 8029 0002 98BE E784 CAE1 BD6C ACD6 F2AE
0003 NOP              7530 --- 8029 0003 98BE E784 CAE1 BD6C ACD6 F2AE
0003 NOP              7530 --- 8029 0004 98BE E784 CAE1 BD6C ACD6 F2AE
0003 NOP              7530 --- 8029 0005 98BE E784 CAE1 BD6C ACD6 F2AE
```

どうですか。ちゃんとGR1の値は1～5まで1つづつ動いていることが分かりますね。

なお、このプログラムでは、「ADDA GR1,=1」と「CPA GR1,=5」の部分にリテラルを用いています。しかし、リテラルは、その値を主記憶装置から取り出して参照します。つまり、その演算ごとに主記憶装置へのアクセスが発生します。そこで、もし汎用レジスタに余裕があれば、これらの値(リテラルで定義した1や5という数値)は他の汎用レジスタに入れておいた方がよいでしょう。

比較する値を他の汎用レジスタに入れたプログラム(LOOP2.casl)を以下に示します。このプログラムではGR2に1の値、GR4に5の値を入れてあります。

```
LOOP2   START
        LAD     GR2,1       GR2←1(定数)、カウントアップ用
        LAD     GR4,5       GR4←5(最終値)
        XOR     GR1,GR1     GR1←0, GR1をカウンタリセット
LOOP    ADDA    GR1,GR2     GR1(カウンタ)をカウントアップ
        NOP                 ;       ループの処理
        CPA     GR1,GR4     GR1:GR4(=5)比較
        JMI     LOOP        GR1 < GR4のときLOOPへ
        RET
        END
```

この方が、主記憶装置へのアクセス回数が少なくなり、その分効率がよいと言えるでしょう。

なお、このようにすると比較演算命令「CPA GR1,GR4」を間違って「CPA GR4,GR1」と書いてしまうことも考えられます。しかし、表9-2に示すように、この2つの比較命令の結果は異なりますので注意してください。

**表9-2 「CPA GR1,GR4」と「CPA GR4,GR1」の違い**

| GR1の値 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| CPA GR1, GR4 | -S- | -S- | -S- | -S- | -S- | --Z | --- | --- |
| CPA GR4, GR1 | --- | --- | --- | --- | --- | --Z | -S- | -S- |

表の説明 GR4の値は5とする

なお、ループ基本アルゴリズム1を一般化して、GR1の値をmからnまで1つずつ増加させるには次のようにします。

```
        GR1← [ m-1 ]
LOOP    GR1←GR1+1(カウンタ更新命令)
        ループの処理
        CPA   GR1,=[ n ]     GR1と[ n ]を比較する
        JMI   LOOP
```

ループ基本アルゴリズム2
(GR1=m,m＋1,m＋2,…,n－1,n)

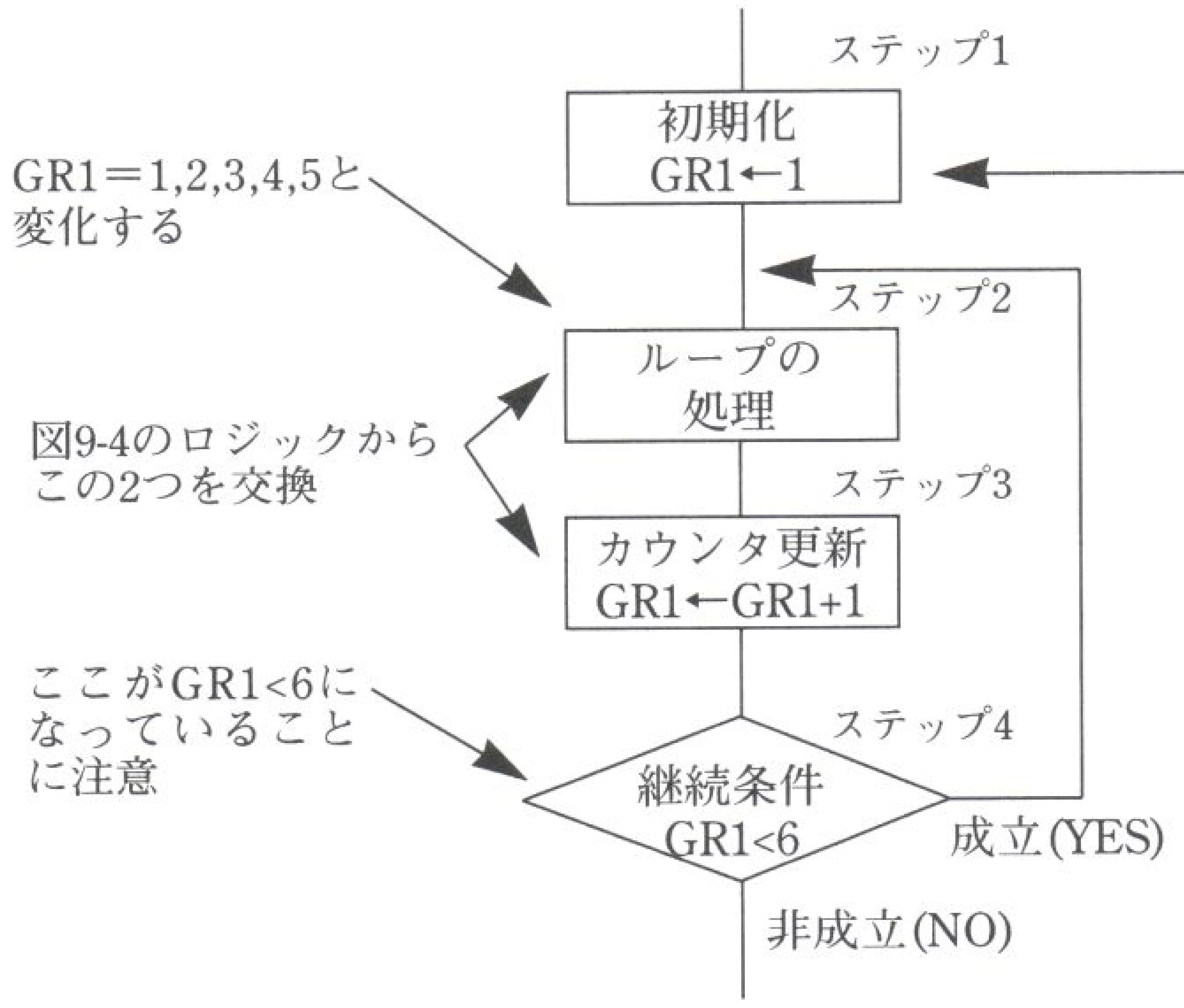
[ ] の中には実際の数値が定数として入ります。例として、－3から4まで動かすプログラム(LOOP1G.casl)を紹介しましょう。

```
LOOP1G  START
        LAD     GR1,-4      GR1←(-3-1), GR1をカウンタセット
LOOP    ADDA    GR1,=1      GR1(カウンタ)をカウントアップ
        NOP               ;     ループの処理
        CPA     GR1,=4      GR1:GR4(=4)比較
        JMI     LOOP        GR1 < 4のときLOOPへ
        RET
        END
```

リテラルは必要に応じて汎用レジスタに変えてください。また、たとえば、カウンタの値を1,3,5,7,9と2つずつ動かすには、カウンタの値を+2します(演習9-21(1)参照)。

(2) 方法2：GR1を1,2,3,4,5まで動かす

次に図9-4に示したアルゴリズム(9.1)の「ループの処理」と「カウンタ更新」の順序を入れ替えたプログラムを示します。




判定の様子

| GR1の値 | カウンタ更新後のGR1の値 | 継続条件 | 判定 |
|---|---|---|---|
| 1 | 2 | 2<6 | YES |
| 2 | 3 | 3<6 | YES |
| 3 | 4 | 4<6 | YES |
| 4 | 5 | 5<6 | YES |
| 5 | 6 | 6<6 | NO |

**図9-5　アルゴリズム(9.2)：GR1を1,2,3,4,5と動かすループ**

ループの進行によって、フラグレジスタがどのように変化するかを表9-3に示しました。カウンタ全体が1つずれているので、継続条件の判定も1つずれてGR1<6になっていることに注意してください。

**表9-3　アルゴリズム(9.2)のループの動作の様子**

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ステップ1 | 1 | | | | | | | | | | | | | | | | ←ステップ実行の様子 |
| ステップ2 | | 2 | | | 2 | | | 2 | | | 2 | | | 2 | | | そのステップが実行されたことを |
| ステップ3 | | | 3 | | | 3 | | | 3 | | | 3 | | | 3 | | そのステップ番号で示す |
| ステップ4 | | | | 4 | | | 4 | | | 4 | | | 4 | | | 4 | |
| GR1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | |
| OSZ | ? | | | -S- | | | -S- | | | -S- | | | -S- | | | -Z- | CPA　GR1,=6の結果 |

―― ループの継続 ――→　└→ループの終了

このロジックの一般形を示します。

```
       GR1←1
LOOP   ループの処理
       GR1←GR1+1(カウンタ更新命令)
       CPA   GR1,=6 GR1と6を比較する
       JMI   LOOP
```

ループ基本アルゴリズム3
(GR1=1,2,3,4,5)

このロジックによるプログラム例(LOOP3.casl)を次に示します。

```
LOOP3   START
        LAD     GR1,1       GR1←1
LOOP    NOP                ;ループの処理
        ADDA    GR1,=1      GR1(カウンタ)をカウントアップ
        CPA     GR1,=6      GR1:6 比較
        JMI     LOOP        GR1 < 6のときLOOPへ
        RET
        END
```

この例のように、ループ中のカウンタの更新の場所やループ処理の場所を少しでも変更すると、プログラム全体のタイミングがずれて、条件判断の処理の仕方が異なってきますので、注意が必要です。特に、CASL IIでプログラムを組む場合、あるいは、情報処理技術者試験の問題を解く場合には、さまざまなループの考え方に慣れておく必要があります。というのは、このようなタイミングの問題は私たち人間が最も不得意とする分野なのです。情報処理技術者試験のときには時間が限られていますから、その限られた時間の中でループロジックを判別する必要があります。これには訓練が必要です。

ところで、この方法で、GR1をm〜n(m≦n)の範囲でループするには、次のようにします。リテラル表現は、必要に応じて汎用レジスタに変えてください。

```
       GR1← [ m ]
LOOP   ループの処理
       GR1←GR1+1   (カウンタ更新命令)
       CPA   GR1,=[ n+1 ]  GR1と[ n+1 ]を比較する
       JMI   LOOP
```

ループ基本アルゴリズム4
(GR1=m,m+1,m+2,…,n−1,n)

たとえば、0〜7まで繰り返すには、次のようにします。

```
LOOP3G  START
        XOR     GR1,GR1     GR1←0
LOOP    NOP                ;     ループの処理
        ADDA    GR1,=1      GR1(カウンタ)をカウントアップ
        CPA     GR1,=8      GR1:8 (=7+1)比較
        JMI     LOOP        GR1 < 8のときLOOPへ
        RET
        END
```

この例の「0〜nまで繰り返す」というループ処理は、第11章で学習するテーブル処理を実現する際に必要になる考え方ですので、ぜひマスターしておいてください。

また、GR2の値を−3から5まで繰り返すには、次のようにします(LOOP3M.casl)。

```
LOOP3M   START
         LAD      GR2,-3    GR2 ← -3
LOOP     NOP                 ;ループの処理
         ADDA     GR2,=1    GR2(カウンタ)をカウントアップ
         CPA      GR2,=6    GR2:6 (=5+1)比較
         JMI      LOOP      GR2 < 6 のとき LOOP へ
         RET
         END
```

## 演習問題9-2-1

(1) 次のプログラムは、GR1を1,3,5,7,9と動かすプログラム(一部)です。空欄を埋めてください。

```
          LAD     GR2, [  1  ]
          LAD     GR4,9
          LAD     GR1, [  2  ]
LOOP      ADDA    GR1,GR2
          NOP
          CPA     GR1, [  3  ]
       [  4  ]   LOOP
```

(2) プログラムLOOP1とLOOP3を次の指示に従って変更して下さい。変更後の名前は、LOOP1をLOOP1Aに、LOOP3をLOOP3Aとして下さい。

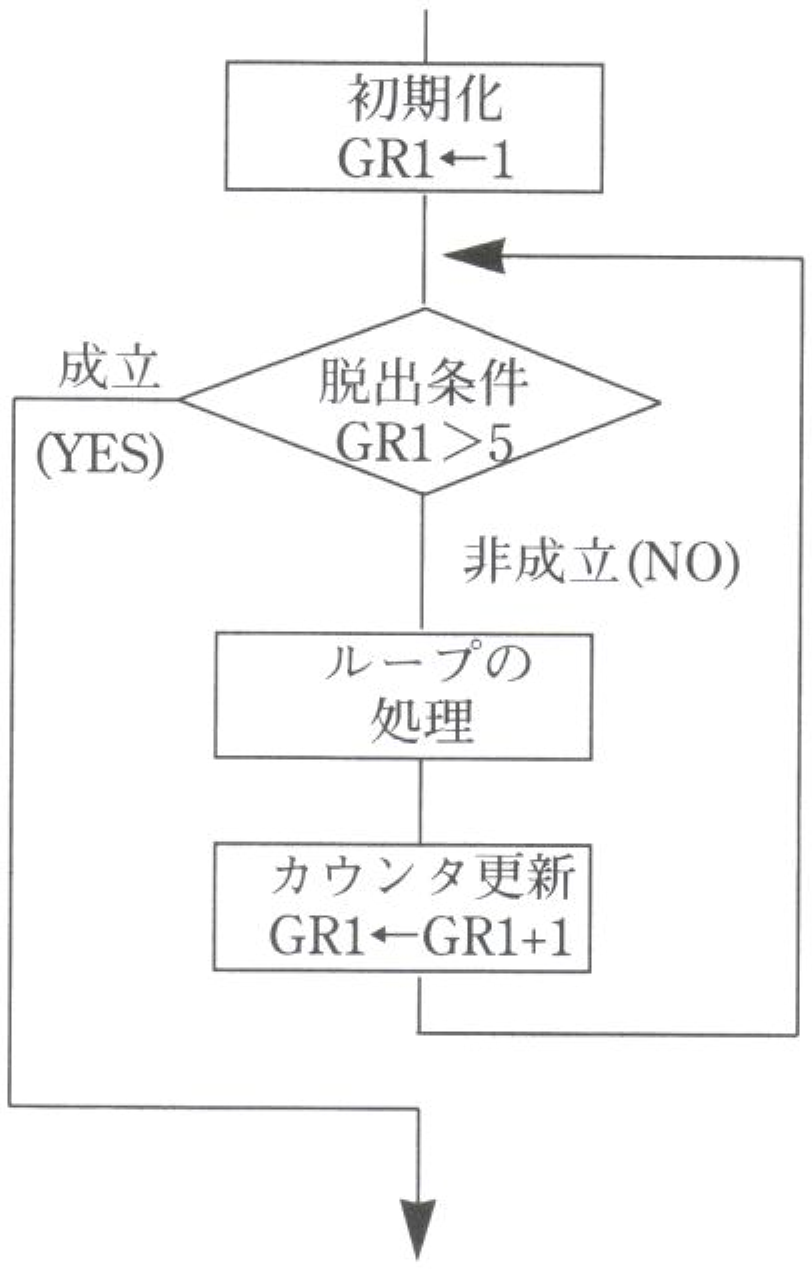
指示：

処理の最後にフラグレジスタのZFが立っていることに着目して、ループの継続条件として「ZFが立っていない間」と考えたプログラムを作成してください。これは、JNZ命令で実現できます。

(3) LOOP3をリテラルを用いない形式に改めなさい(LOOP3B.casl)。

### 9.2.2 GR1を1,2,3,4,5まで動かす：脱出条件処理

図9-2のパターン2を用いて、GR1を1,2,3,4,5まで動かすプログラムについて考えてみましょう。これは、次のアルゴリズムになります。




判定の様子

| GR1の値 | 脱出条件 | 判 定 |
|---|---|---|
| 1 | 1＞5 | NO |
| 2 | 2＞5 | NO |
| 3 | 3＞5 | NO |
| 4 | 4＞5 | NO |
| 5 | 5＞5 | NO |
| 6 | 6＞5 | YES |

**図9-6　アルゴリズム(9.3)：GR1を1,2,3,4,5と動かすループ**

「CPA　GR1,＝5」の判定の様子を表9-4にまとめます。上記のフローチャートから分かるように、この場合には、脱出条件が成立したらループを終了するわけですから、GR1が5より大きくなったらループを脱出するようにします。この判定は、まず、「CPA　GR1,＝5」とし、次にSFもZFも立たないときにジャンプするJPL命令で行うことができます。

**表9-4　判定の様子**

| GR1の値 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| CPA GR1, =5 | -S- | -S- | -S- | -S- | --Z | --- | --- |
| JPLの動作 | × | × | × | × | × | ○ | ○ |

×：ジャンプしない、　○：ジャンプする

このプログラムの概要は次のようになります。

```
        GR1←1
LOOP    CPA  GR1,=5
        JPL  EXITLOOP
        ループの処理
        GR1←GR1+1    (カウンタ更新命令)
        JUMP LOOP
EXITLOOP
```

ループ基本アルゴリズム5
(GR1=1,2,3,4,5)

このロジックによるプログラム例(LOOP4.casl)を次に示します。

```
LOOP4     START
          LAD     GR1,1
LOOP      CPA     GR1,=5
          JPL     EXITLOOP
          NOP        ; ループの処理
          ADDA    GR1,=1  GR1(カウンタ)をカウントアップ
          JUMP    LOOP
EXITLOOP  RET
          END
```

なお、このアルゴリズムで、GR1をm〜nに動かしたい場合には、次のようにします。

```
         GR1←[ m ]
LOOP     CPA  GR1,=[ n ]
         JPL  EXITLOOP
         ループの処理
         GR1←GR1+1(カウンタ更新命令)
         JUMP LOOP
EXITLOOP
```

ループ基本アルゴリズム6
(GR1=m,m+1,m+2,…,n−1,n)

このアルゴリズムは、m,nの値がプログラム中でもそのまま使えるので分かりやすいです。

## 演習問題9-2-2

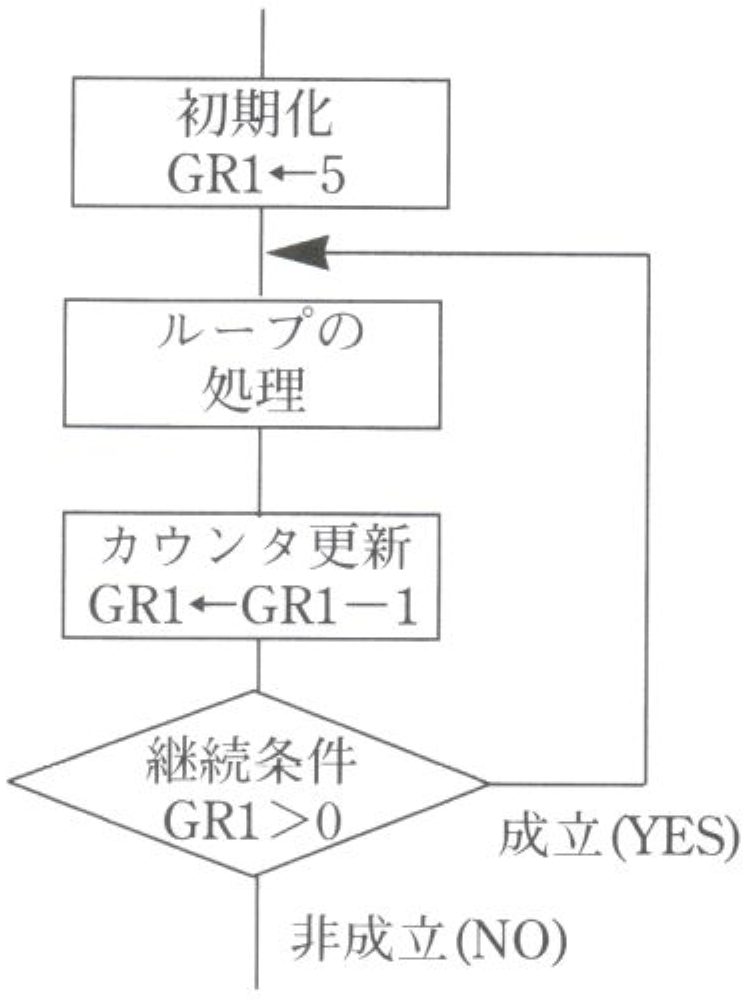
次のプログラムは、LOOP4のプログラムのGR1のカウントアップの部分とNOP(ループの処理)の部分を交換したものです。ループの処理(NOP)の部分を通過するたびに、GR1の値が1〜5まで動くように空欄を埋めてください。

```
LOOP4Q1   START
          XOR     GR1,[  1  ]    GR1←0
LOOP      CPA     GR1,=5
          [  2  ] EXITLOOP
          ADDA    GR1,=1         GR1(カウンタ)をカウントアップ
          NOP        ; ループの処理
          JUMP    LOOP
EXITLOOP  RET
          END
```

### 9.2.3 GR1を5,4,3,2,1まで動かす：カウントダウン処理

今度は、汎用レジスタの値をカウントダウンしていく方法について学習しましょう。カウントダウンの場合には、カウンタが0になることを利用してループを制御する方法がよく用いられます。その場合のカウンタの最後の値は、0か1になります。しかし、この方法は、カウントアップ処理よりも単純なアルゴリズムでプログラムを作成することができるので、よく用いられます。

基本パターンの構造は次のようになっています。




判定の様子

| GR1の値 | 継続条件 | 判 定 |
| --- | --- | --- |
| 5 | 5＞0 | YES |
| 4 | 4＞0 | YES |
| 3 | 3＞0 | YES |
| 2 | 2＞0 | YES |
| 1 | 1＞0 | YES |
| 0 | 0＞0 | NO |

**図9-7　アルゴリズム(9.4)：GR1を5,4,3,2,1と動かすループ**

これは、次のような単純な流れになっています。

```
        GR1←5
LOOP    ループの処理
        GR1←GR1-1
        JPL LOOP      GR1>0のときLOOPへ(GR1=5,4,3,2,1のとき)
```

ループ基本アルゴリズム7
(GR1=5,4,3,2,1)

こんなに単純に書けるのは、カウントダウンしたときのフラグレジスタの値を直接利用しているからです。実際のプログラム(LOOP5.casl)を次に示します。

```
LOOP5       START
            LD        GR1,=5   GR1←5, GR1はカウンタ
            LOOP      NOP      ; ループの本体
            SUBA      GR1,=1   GR1(カウンタ)をカウントダウン
            JPL       LOOP     count > 0 のときLOOPへ
            RET
            END
```

もし、GR1の値をn, n−1,…, 2,1,0まで動かしたいなら、次のように分岐命令を組み合せます。

```
        GR1←5
LOOP    ループの処理
        GR1←GR1-1
        JPL LOOP      GR1>0のときLOOPへ(GR1=5,4,3,2,1のとき)
        JZE LOOP      GR1=0のときLOOPへ(GR1=0のとき)
```

ループ基本アルゴリズム8
(GR1=n,n−1,n−2,…,2,1,0)

「GR1←GR1−1」(実際の命令では、「SUBA GR1,=1」など)で変化したフラグレジスタの内容は、分岐命令(JPL)を実行しても変化しません。つまり、「GR1←GR1−1」で変化したフラグレジスタは、分岐命令(JZE)でもそのまま使えます。したがって、2つめの分岐命令(JZE)でGR1がゼロのときもル

ープさせるようにしているのです。

ちなみに、この2つの分岐命令(JPL, JZE)の順番を逆転させてもプログラムは正しく動作しますが、効率が若干低下します。なぜなら、GR1のほとんどの判断は、「JPL」で行われ、「JZE」で判断されるのは最後の1回だけだからです。もし、先に「JZE」の判断を行うと、この判断処理がすべてのループの中で行われてしまい、余分な機械語を実行することになってしまいます。些細なことのようですが、ループの中の機械語命令は、何度も(場合によっては数万回)実行されることがあるので、このような配慮が必要になります。

ところで、判断処理をどうしても一つにしたいのなら、次のようにすることもできます。

```
         GR1← [ n ]
LOOP     JMI    NEXT
         ループの処理
         SUBA   GR1,=1
         JUMP   LOOP
NEXT     次の命令
```

ループ基本アルゴリズム 9
(GR1=n,n−1,n−2,…,2,1,0)

```
         GR1 ← [ n+1 ]
LOOP     SUBA   GR1,=1
         ループの処理
         CPA    GR1,=0
         JPL    LOOP
```

ループ基本アルゴリズム 10
(GR1=n,n−1,n−2,…,2,1,0)

実際のプログラムを示します。

```
LOOP5Z1     START
            LAD       GR1,5  GR1←5
LOOP        JMI       NEXT
            NOP
            SUBA      GR1,=1
            JUMP      LOOP
NEXT        RET
            END
```

```
LOOP5Z2     START
            LAD       GR1,6
LOOP        SUBA      GR1,=1
            NOP
            CPA       GR1,=0
            JPL       LOOP
            RET
            END
```

また、m, m−1, m−2,…, n+1, nとカウントダウンするアルゴリズムは、たとえば、次のようになります。

```
        GR1 ← [ m+1 ]
LOOP    SUBA GR1,=1
        ループの処理
        CPA  GR1,=[n]
        JPL  LOOP
```

ループ基本アルゴリズム 11
(GR1=m,m−1,m−2,…,n+1,n)

## 演習問題9-2-3

プログラムLOOP5で、GR1をカウントダウンするために、「LAD GR1,−1,GR1」としたのではだめですか。

## 9.3 集計処理

### 9.3.1 1〜Nの和の集計

> **例題9-1**
>
> Nに格納された値を用いて、S=1+2+3+4+…+Nを求めるプログラムを作りなさい。

このプログラムは、アルゴリズム9.1〜9.4の単純な応用と言えるでしょう。集計処理の基本的な考え方について説明します。集計処理を行うには、その集計結果を保持する領域あるいは汎用レジスタを用意します。通常は効率の良い汎用レジスタを用います。ここでは、集計結果を格納する場所としてGR3を用いましょう。さて、集計処理の最初の処理として、GR3をゼロクリアします。そのためには、XOR命令で次のようにします。

```
XOR GR3,GR3        GR3 [    0    ]
```

そして、ここに、1からNまでの値を次々に加えていけばよいのです。

```
GR3←GR3+1          GR3 [    1    ]
GR3←GR3+2          GR3 [    3    ]
GR3←GR3+3          GR3 [    6    ]
    ∫
GR3←GR3+N          GR3 [1〜Nまでの和]
```

ループの中でレジスタGR1の値を1からNまで動かすようにすれば、上記の加算命令は「GR3←GR3+GR1」すなわち「ADDA　GR1,GR3」となります。

なお、ここでは集計処理をカウントアップで実現する方法とカウントダウンで実現する方法の2つを示すことにしましょう。

#### (1) カウントアップで処理する方法

アルゴリズム9.1を応用したプログラム(SUM1.casl)を次に示します。なお、汎用レジスタが十分余っているので、カウントアップする際にリテラル(ADDA GR1,=1)を使わないようにしています。また、比較演算命令で領域Nの値と常に比較する(CPA GR1,N)とすると、そのたび毎に主記憶装置へデータのアクセスが発生するので(Nの値を取り出す必要があるため)、レジスタGR4にNの値をロードして用いています。

```
SUM1    START
        XOR     GR1,GR1     GR1←0, GR1をカウンタリセット
        LD      GR4,N       GR4←N, GR4はリミット
        LAD     GR2,1       GR2←1(定数)
        XOR     GR3,GR3     GR3←0(合計値)
LOOP    ADDA    GR1,GR2     GR1(カウンタ)をカウントアップ
        ADDA    GR3,GR1     GR3 = GR3 + GR1(カウンタ)
        CPA     GR1,GR4     GR1(汎用レジスタ)
        JMI     LOOP         if GR1 < N then goto LOOP
        ST      GR3,S       GR3→S
        RET
S       DS      1
N       DC      10
        END
```

このプログラムを本書に付属のCASLDVにかけて、トレースして、レジスタGR1とGR3の値が変化する様子を観察してください。

### (2) カウントダウンで処理する方法

アルゴリズム9.4を応用したプログラム(SUM2.casl)を次に示します。

```
SUM2    START
        LD      GR1,N       GR1←N, GR1はカウンタ
        LAD     GR2,1       GR2←1(定数)
        XOR     GR3,GR3     GR3←0(合計値)
LOOP    ADDA    GR3,GR1     GR3 = GR3 + GR1(カウンタ)
        SUBA    GR1,GR2     GR1(カウンタ)をカウントダウン
        JNZ     LOOP         if count ≠ 0 then goto LOOP
        ST      GR3,S       GR3→S
        RET
S       DS      1
N       DC      10
        END
```

## 演習問題9-3-1(a)

(1) GR3をゼロクリアするのにXOR命令を用いています。他にどんな命令でゼロクリアできましたか。

(2) SUM1のプログラムをリテラル(ADDA GR1,=1)を使って書いてください。また、領域Nの値も「CPA　GR1,N」としてそのまま使ってください(SUM1A)。そして、なぜ、この方法が効率が悪いか説明してください。

(3) SUM1を図9-5アルゴリズム9.2(LOOP3→SUM1B)、図9-6アルゴリズム9.3(LOOP4→SUM1C)で書き直してください。

**例題9-2**

N(偶数)に格納された値を用いて、S=2＋4＋6＋…＋Nを求めるプログラム(SUM3.casl)を作りなさい。なお、Nが偶数かどうかチェックする必要はありません(チェック処理を入れる問題は演習問題9-3-1(b)を参照)。

この例題はカウントダウンのアルゴリズムで作成してみましょう。この場合には、カウンタGR1をN(偶数)から2個ずつ0になるまでカウントダウンすればよいでしょう。

```
SUM3    START
        LD      GR1,N       GR1←N, GR1はカウンタ
        LAD     GR2,2       GR2←2(定数)
        XOR     GR3,GR3     GR3←0(合計値)
LOOP    ADDA    GR3,GR1     GR3 = GR3 + GR1(カウンタ)
        SUBA    GR1,GR2     GR1(カウンタ)を2つカウントダウン
        JNZ     LOOP        if count ≠ 0 then goto LOOP
        ST      GR3,S       GR3→S
        RET
S       DS      1
N       DC      10
        END
```

上記のプログラムから分かるように、プログラムSUM3は、プログラムSUM2の3行目を次のように変えただけです。

| LAD　GR2,1 | → | LAD　GR2,2 |
|---|---|---|
| (プログラムSUM2) | | (プログラムSUM3) |

# 演習問題9-3-1(b)

プログラムSUM3に、Nが奇数ならばSにゼロをセットし終了するプログラムを付け加えたプログラム(SUM3A)を示します。Nが偶数なら、そのまま2+4+…+Nを求めてSにセットします。空欄に必要なプログラムを記述してください。ただし、[ 2 ]には2種類の分岐命令が答えとして入ります。両方とも答えて下さい。

```
SUM3A      START
           LD        GR1,N        GR1←N, GR1はカウンタ
           LD        GR2,GR1      GR2←GR1
           AND       [ 1 ],#0001  最下位ビットを抽出
           [  2  ]   ODDVALUE     奇数ならODDVALUEへ
           LAD       GR2,2        GR2←2(定数)
           XOR       GR3,GR3      GR3←0(合計値)
LOOP       ADDA      GR3,GR1      GR3 = GR3 + GR1(カウンタ)
           SUBA      GR1,GR2      GR1(カウンタ)を2つカウントダウン
           JNZ       LOOP         if count ≠ 0 then goto LOOP
SETV       ST        GR3,S        GR3→S
           RET
ODDVALUE   XOR       GR3,[  3  ]
           [        4        ]
S          DS        1
N          DC        9
           END
```

## 9.3.2 集計値のオーバーフロー

**例題9-3**

1+2+3+…+Nの値が$2^{16}-1$=65535を越えない最大のNの値を求めなさい。

ループを用いて1+2+3+…+Nを論理加算で求めていき、オーバーフローした時点でループを抜け、その一つ前のカウンタの値を求めれば良いでしょう。この説明が分かりにくい場合には、問題を小さくして実際の具体的な数を用いて考えます。たとえば、「S=1+2+3+…+Nの値が12を越えない最大のNの値」について考えます。Nの値とSの値を表(次ページ)にしてみると次のようになり、S=15、すなわち、最初に12を越えたときのNの値が5ですからそれよりも1少ない4が求める答えになります。この小さな問題での「12を越えた」という判定がちょうどこの例題における「オーバーフローした」という判定に相当します。

```
N  S
1  1              1
2  1+2            3
3  1+2+3          6
4  1+2+3+4       10
5  1+2+3+4+5     15      ← 12を越えた
```

プログラム(SUM4.casl)を以下に示します。

```
SUM4        START
            XOR      GR1,GR1      GR1←0, GR1をカウンタリセット
            LAD      GR2,1        GR2←1(定数)
            XOR      GR3,GR3      GR3←0(合計値)
LOOP        ADDL     GR1,GR2      GR1(カウンタ)をカウントアップ
            ADDL     GR3,GR1      GR3 = GR3 + GR1(カウンタ)
            JOV      EXITLOOP     オーバーフローならループを終了
            JUMP     LOOP         LOOPへジャンプ
EXITLOOP    SUBL     GR1,GR2
            ST       GR1,N        GR3→N
            RET
N           DS       1
            END
```

これを実行すると、Nには361が格納されます。$\sum_{k=1}^{361} k=65341$、$\sum_{k=1}^{362} k=65703$ですから、正しく計算されていることが分かります。

## 演習問題9-3-2

1＋2＋3＋….＋Nの値が$2^{15}-1=32767$を越えない最大のNの値を求めるプログラムを作りなさい(SUM4A.casl)。

## 9.4 乗算・除算

COMET IIには、乗算と除算の命令がありません。$2^k$倍や$2^k$で割る方法はすでに学習しましたが、任意のデータを乗算したり除算するにはループを使うしか方法がありません。第10章で学習するビット操作を使えば、任意のデータを効率よく計算(乗算・除算)するプログラムを作れますが、ここでは、乗算を加算の繰り返しに、除算を減算の繰り返しに置き換えた簡単なプログラムを学習しましょう。

### 9.4.1 加算による乗算プログラム

**例題9-4**

A(≧0)、B(≧0)のデータに対して、C←A×Bを計算するプログラム(MULT.casl)を作りなさい。ただし、データがゼロ以上であることを確認するエラーチェックは省いてよい。

A×Bの計算手順は$\underbrace{A+A+A+\cdots+A}_{B回加える}$です。したがって、次のプログラムになります。

```
MULT      START
          XOR     GR1,GR1     GR1←0, 解を格納するレジスタ
          LD      GR2,B       GR2←B, B回ループするカウンタ
          JZE     FINISH      B=0なら解もゼロ
          LD      GR3,A       GR3←A
          LAD     GR4,1       GR4←1, 定数、カウントダウン用
MLTLOOP   ADDA    GR1,GR3     GR1←GR1+GR3 ,積を求める
          SUBA    GR2,GR4     カウンタのカウントダウン
          JPL     MLTLOOP     カウンタが1以上ならMLTLOOPへ
FINISH    ST      GR1,C       GR1(解)→C
          RET
A         DC      125
B         DC      148
C         DS      1
          END
```

このプログラムでは、レジスタGR1にAの値をB回加えています。ループのカウンタにはレジスタGR2を用います。つまり、レジスタGR2に最初Bの値をセットして、この値をカウントダウンしてB回ループを繰り返します。

## 演習問題9-4-1

(1) プログラムMULTにおいて、A=3,B=100の計算時に100回ループするのは無駄です。そこで、A,Bの値を調べ、少ない方をループ回数とするプログラム(MULTA.casl)を作成してください。

(2) プログラム(MULTA)に、オーバーフロー処理を追加して下さい。つまり、乗算結果がオーバーフローした場合、結果を正の最大値にして、領域OVFに1をセットし、オーバーフローしない場合、領域OVFに0をセットするプログラム(MULTB.casl)を作成してください。

### 9.4.2 減算による除算プログラム

**例題9-5**

A(≧0)、B(>0)のデータに対して、C←A÷Bを計算するプログラム(DIV.casl)を作りなさい。また、その時の余りをRに格納しなさい。ただし、Bがゼロのときはエラーチェックを行い、'zero divide'と表示してください。しかし、それ以外のデータのエラーチェックは省いてよい。

AからBを何回引けるかチェックし、引けた回数が商、最後に残った値が余りになります。

```
DIV      START
         LD     GR3,B          GR3←B
         JZE    ERROR          もしGR3=0ならERRORへジャンプ
         XOR    GR1,GR1        GR1(商)←0
         LD     GR2,A          GR2←A(ここからどんどん引いていく)
         LAD    GR4,1          GR4←1
DIVLOOP  CPA    GR2,GR3        GR2とGR3(=B)を比較
         JMI    FINISH         GR2からGR3(=B)が引けなくなったらFINISH
         SUBA   GR2,GR3        GR2←GR2-GR3(=B)
         ADDA   GR1,GR4        GR1←GR1+GR4(=1)
         JUMP   DIVLOOP        DIVLOOPへジャンプしてループを形成
FINISH   ST     GR1,C          GR1→C(商)
         ST     GR2,R          GR2→R(余り)
         RET                    ; プログラムの終了
ERROR    OUT    ERRMSG,MSGLEN  エラーメッセージの出力
         RET                    ; プログラムの終了
A        DC     1253
B        DC     64
C        DS     1              商
R        DS     1              余り
ERRMSG   DC     'zero divide'
MSGLEN   DC     11
         END
```

除算は、前述のように減算を用いて行うことができます。たとえば、14を5で割るには次のようにします。

　　　　1回目の減算　14 − 5 = 9
商 →　2回目の減算　 9 − 5 = 4 ← 余り

上に示すように、12から5を2回引くことができます。したがって、商は2となり、残った4が余りとなります。

プログラムDIVでは、A÷Bを計算するのに、まず、GR3にBの値を入れます。もし、Bがゼロならエラーメッセージを出力します。次にAの値をGR2に入れて、ここから次々にBの値(すなわち、GR3)を引いていきます。この回数をGR1に入れます。このようにすることで、ループ終了時にはGR1に商が、GR2に余りが入っていることになります。

なお、エラーメッセージの出力はOUT命令を用いていますが、出力領域としてERRMSGおよび出力文字数にMSGLENを用意して「OUT　ERRMSG,MSGLEN」としています。しかし、これをリテラルを用いて次のように1行だけで記述することもできます(＊1)。

```
OUT   ='zero divide',=11
```

## 演習問題9-4-2

(1) 秒数がSECONDに格納されています。これを、時間・分・秒に変換して、それぞれ、時間(HOUR)、分(MINUTE)、秒(SECOND)に格納するプログラム(CNVTIME.casl)を作りなさい。

(2) Aを5で割って、その答えを小数点第1位まで求め、整数部をX、小数部をYに格納するプログラム(DIV5.casl)を作りなさい。

[ヒント] 5で割って余りが0なら小数点第1位も0、1なら小数点第1位は2、2なら小数点第1位は4、...です。

(＊1) 実は、OUT命令にリテラルが使用できるとは、CASL IIの仕様にはどこにも記載されていません。しかしOUT命令はマクロ展開によって4.6節で示すように別の機械語命令に展開されることになっています。そこで、CASLDVは、OUT命令へのリテラルの記述を可能にしています。

# 総合問題9

**[演習9-1]** フィボナッチ数列(Fibonacci sequence)は、次の式で定義される単純な数列です。

$$F_0=0,\ F_1=1,\ F_{i+1} = F_{i-1} + F_i$$

この数列の第20項めを計算して領域F20に格納するプログラム(FIB20.casl)を作成してください。

**[演習9-2]** どのような自然数も、次の手順を限りなく行うと、最後は必ず1になることが知られています。

・もしその数が奇数なら3倍して1を加える
・もしその数が偶数なら2で割る

たとえば、自然数を6とすると、

6は偶数なので2で割って3
3は奇数なので3×3+1=10
10は偶数なので2で割って5
5は奇数なので3×5+1=16
16は偶数なので2で割って8
8は偶数なので2で割って4
4は偶数なので2で割って2
2は偶数なので2で割って1 ——→ 1になった！

領域Nに格納された任意の自然数に対してこの処理を行い、1になったら終了するプログラムを作ってください(LPCHK.casl)。

**[演習9-3]** ファイルを入力して、そのまま出力するプログラム(FILEPR. cas1)を作成せよ。

# 第10章 ビット操作

ビット操作は、効率のよいプログラムを書くことが必要なアセンブラ言語では必須のテクニックです。本章では、多くの例題を通して、ビット操作の処理の方法について解説していきます。

また、本章の演習問題には、かなり本格的な例題が含まれていますので、ぜひ独力で解いて力をつけてください。

## 10.1 ビット操作とは

データは、0と1の2進数から構成されるということはすでに学習しています。**ビット操作**は、この0と1を直接扱う操作のことです。すでに、6.3節「論理演算命令」で、AND命令、OR命令、XOR命令を学習しています。本章の狙いは、この命令を使って、具体的なビット操作のプログラミングを学習することです。

いくつかの基本的なビット操作の処理については、表10-1に示すようにすでに学習しています。

**表10-1 すでに学習しているビット操作(一部除く)**

| 場所 | 処理内容 | 例 | 例の説明 |
|---|---|---|---|
| 7.1.2 | ゼロクリア | XOR GR1,GR1 | GR1の内容をゼロにする |
| 5.2 | $2^k$倍 | SLA GR1,2 | GR1の内容を4倍する |
| 5.2 | $2^k$で割る | SRA GR1,3 | GR1の内容を8で割る |
| 表6-7 | 符号チェック | AND GR1,=#8000 | 最上位ビットを取り出す(ただし、この方法よりもSFをチェックする方が効率的) |
| 表6-7 | 奇遇チェック | AND GR1,=1 | 最下位ビットを取り出す |
| 表6-7 | $2^k$の倍数のチェック | AND GR1,=7 | 8の倍数の時に0、8の倍数でないとき1〜7になる |
| 表6-10 | ビット反転 | XOR GR1,=#FFFF | $FFFF_{(16)}$とXORを取る |
| 表6-7 | ビット抽出 | AND GR1,=#0008 | 第3ビット目を取り出す |
| 表6-8 | ビット設定 | OR GR1,=#0010 | 第4ビット目を1に設定 |
| 8.2.1 | 補数を求める | XOR GR1,=#FFFF<br>ADDA GR1,=1 | ビットを反転して<br>1を加える |
| 表6-9 | 英大文字→小文字変換 | OR GR1,=#0020 | GR1に'A'が入っていれば'a'になる |
| 表6-9 | 英小文字→大文字変換 | AND GR1,=#FFDF | GR1に'a'が入っていれば'A'になる |
| | トグルスイッチ | XOR GR1,=1 | GR1が1のとき0になり、0のとき1になる |
| 10.4 | 16回ループする | SLL GR1,1<br>JOV FINISH | GR1に1を入れておき、16回SLLしたらOFが立つことを利用 |

## ポイント

表10-1のビット操作テクニックは、必ず理屈を理解して覚えることが大切！

## 演習問題10-1

(1) 次の各処理を行うプログラムの空欄の部分に当てはまる語句を入れてください。

| 処理 | 命令 | オペランド |
|---|---|---|
| (a) GR4の値を8倍する | SLA | GR4, [ 1 ] |
| (b) GR4の値を256倍する | SLA | GR4, [ 2 ] |
| (c) GR7の値を0にする | XOR | GR7, [ 3 ] |
| (d) GR1の内容が4の倍数なら | AND | GR1, [ 4 ] |
| B4へジャンプする | [ 5 ] | B4 |
| (e) GR5の値を1/16倍にする | SRA | GR5, [ 6 ] |
| (f) GR5の値を2で割る | [ 7 ] | GR5, [ 8 ] |
| (g) GR6の各ビットをすべて反転する | [ 9 ] | GR6,=#[ 10 ] |
| (h) GR4の下位8ビットすべてを反転する | XOR | GR4,=#[ 11 ] |
| (i) GR4の上位4ビットすべてを反転する | [ 12 ] | GR4,=#[ 13 ] |
| (j) GR4の上位3ビットすべてを反転する | [ 14 ] | GR4,=#[ 15 ] |
| (k) GR3の7ビット目を0にする | [ 16 ] | GR3,=#[ 17 ] |
| (l) GR3の8ビット目を0にする | [ 18 ] | GR3,=#[ 19 ] |
| (m) GR0の3ビット目を1にする | [ 20 ] | GR0,=#[ 21 ] |

(2) GR4の値を次の表のGR3の値に従って計算することを考えます。たとえば、GR3の値が7なら、GR4を128倍します。ただし、GR3には0～8の数値しか入っていないとします。

| GR3の値 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| GR4の値を | 1倍 | 2倍 | 4倍 | 8倍 | 16倍 | 32倍 | 64倍 | 128倍 | 256倍 |

上記の処理を行う下記のプログラムの空欄を埋めてください。

SLA　GR4,[　　　]

(3) GR4の値を次の表のGR3の値に従って計算することを考えます。たとえば、GR3の値が−2ならGR4を2倍します。ただし、GR3には−3～2の数値しか入っていないとする(やや難)。

| GR3の値 | −3 | −2 | −1 | 0 | 1 | 2 |
|---|---|---|---|---|---|---|
| GR4の値を | 1倍 | 2倍 | 4倍 | 8倍 | 16倍 | 32倍 |

上記の処理を行う下記のプログラムの空欄を埋めてください。

SLA　GR4,[　　　]

# 10.2 ビットのカウント

ビットの個数を求める問題は単純なようで重要です。たとえば、パリティチェックなどの問題は、重要であり、かつ試験問題などに比較的出題されやすいので、この種のテクニックはよく理解しておく必要があります。本節の演習問題として、循環シフトとパリティチェックなどの話題を取り上げていますので、ぜひそちらも参照してください。

**例題10-1**

Aに格納された論理データ(第15ビット目を符号ビットとして扱わない)中の1のビット数を数え、それをCOUNTにセットするプログラム(BCOUNT.casl)を作成してください。

2263というデータについて考えてみましょう。これは、次のビットパターンをしています。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2263 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

これを、右に1ビット論理シフト(SRL)してみましょう。すると、次のようにビット番号0番めのビット1が右に押し出され、オーバーフローフラグが立ちます。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1131 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | → 1 オーバーフローフラグが立つ |

これをさらに次々に右に1ビットづつ論理シフトしていくと次のようになります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 565 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | → 1 オーバーフローフラグが立つ |

↓ (SRL)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 282 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | → 1 オーバーフローフラグが立つ |

↓ (SRL)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 141 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | → 0 オーバーフローフラグが立たない |

↓ (SRL)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 70 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | → 1 オーバーフローフラグが立つ |

↓ (SRL)

上記の結果を見てみると、16回SRLしてオーバーフローフラグの立った回数をカウントすればそれがビット数であることが分かります。

```
BCOUNT    START
          LD     GR1,A
          XOR    GR2,GR2      GR2(オンビット(1)の集計用レジスタ)をゼロにする
          LAD    GR3,16       ループ回数(16回)
          LAD    GR4,1        1(減算用・加算用の定数)
LOOP      SRL    GR1,1        最下位ビットを押し出す
          JOV    BITON        ビットが立っていればBITONへ
CHECK     SUBA   GR3,GR4      GR3(カウンタ)をカウントダウン
          JNZ    LOOP         16回回り切ってなければLOOPへジャンプ
          ST     GR2,COUNT    ループ終了、求められたビット数をCOUNTへ
          RET                 ;プログラムの終了
BITON     ADDA   GR2,GR4      GR2(1のビット数)をカウントアップ
          JUMP   CHECK        CHECKに戻ってループ続行
A         DC     2263
COUNT     DS     1
          END
```

なお、領域Aの中にある1のビットの個数が比較的少ないことが分かっている場合、GR1がゼロであれば、そこで集計処理を終了した方が実行効率が良くなります。これについては本章の総合問題の[演習10-4]を参照してください。

## 演習問題10-2

(1) プログラム(BCOUNT)において、オーバーフローした時の処理(ラベルBITON以下2行)が最後に配置されています。この処理をプログラムの最初に配置したプログラム(BCOUNT1.casl)を作成してください。

| BITON オーバーフロー時の処理 |
|---|
| プログラム本体 |

[ヒント] START命令のオペランドには何を記述したでしょうか。

(2) 次のプログラムは、ビット番号15からビット番号の低い方に向かって調べ、最初に1があるまでの長さを求めるものです。たとえば、次の例ならば、長さは6になります。空欄に適当な字句を入れてください。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 565 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

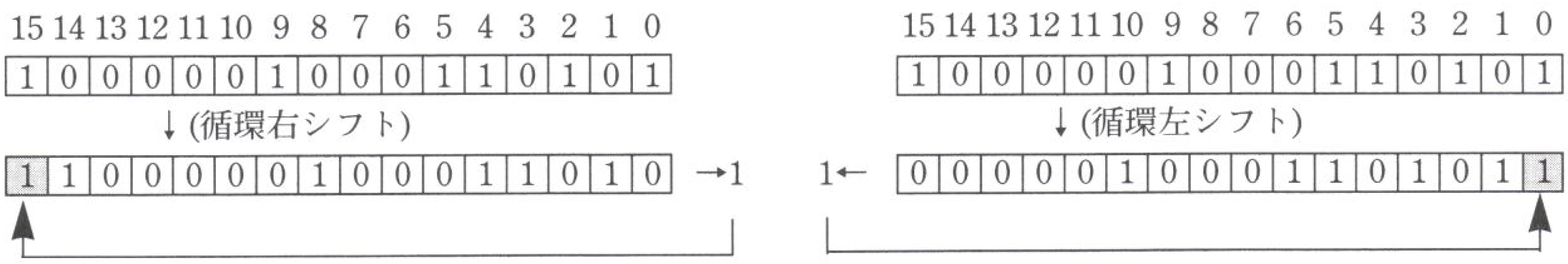
←長さは6→（ビット15〜10）

ヒント：考え方は、このデータを左に1ビットずつ論理シフトしていき、ビットが飛び出た(すなわち、オーバーフローした)時点までのビット0の個数を考えればよいのです。また、このようなプログラムをテスト(デバッグ)するときは、極端な数値を用意す

るとバグが発見しやすいものです。この場合でしたら、FFFF(16)と0000(16)がそれにあたります。FFFF(16)のとき長さは0、0000(16)のとき長さは16となります。この2つのデータはループのカウントの境界をチェックするので、**境界条件**とも呼ばれます。

```
BITCT   START
        LD      GR0,D
        XOR     GR1,GR1
        LAD     GR2,1       定数、カウントアップ用
        LAD     GR3,16      1語のビット数を設定
LOOP    [ 1 ]   GR0,1
        JOV     FOUND
        [     2     ]
        CPA     GR1,GR3
        [ 3 ]   FOUND
        JUMP    LOOP
FOUND   ST      GR1,N
        RET
D       DC      565
N       DS      1
        END
```

(3) 次のプログラムは、1ビットづつ循環シフト処理を行うプログラムです(ただし、循環左シフトのみ)。**循環シフト処理**とは、シフト処理の際に左右に飛び出したビットを逆側の部分に埋め込む処理です。




飛び出したビットが0のときは、0が逆側に設定されます。空欄に適当な字句を入れてください。

```
RSL     START
        LD      GR0,D
        LAD     GR1,1
        SLL     GR0,1
        [ 1 ]   BITON
SET     ST      GR0,RSD
        RET
BITON   [ 2 ]   GR0,GR1
        JUMP    SET
D       DC      #8235
RSD     DS      1
        END
```

(4) 次のプログラムは、偶数パリティのセットを行うプログラムです。このプログラムでは、与えられた語Dに入出力用の文字データが格納されていると仮定します。このDにセットされた文字データに対して偶数パリティを付加することを考えます。入出力用の文字データの場合、第0ビット～第7ビットしか使いません。念のため、第8ビット目から第15ビット目はゼロクリアしておきます。そして、第0ビット～第7ビットの中の1のビットの個数を数えます。もし、それが奇数であれば、第8ビット目に1を立てます。偶数なら何もしません。そして、その結果をPDに格納します。このようにすれば、PDの第0ビットから第8ビット目までの1のビットの個数は常に偶数になります。仮に、どこかで転送エラーが起き、ビット落ち(ビットデータが逆転すること)が発生すると、1のビットの総数が奇数になり、エラーの発生をチェックできます。もちろん、この中で2回ビット落ちが発生すれば、このチェックは無意味になりますが、そもそもビット落ちそのものの発生確率が低いので、これが同時に起こる確率はものすごく低くなります。

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

↓
(この例では0から7の1のビット数が5で奇数)
↓

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

↑
ビットを設定し、ビットの総数を偶数とする

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

↓
(この例では0から7の1のビット数が4で偶数)
↓

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

↑
ビットを設定しない

```
EPARITY START
        LD      GR0,D
        [ 1 ]   GR0,=#00FF    上位8ビットをゼロクリア
        LD      GR3,GR0       GR0は破壊されるため、複写しておく
        XOR     GR1,GR1       1のビットの個数
        LAD     GR2,[ 2 ]     ループ回数
        LAD     GR4,1         定数
;   ビットの数を数える
LOOP    SRL     GR0,[ 3 ]
        JOV     BITON
CHECK   SUBA    GR2,[ 4 ]     カウントダウン
        [ 5 ]   LOOP
; ビットの個数の奇遇をチェック
        [ 6 ]   GR1,GR4
        JPL     ODDBIT        奇数ならジャンプ
SET     ST      GR3,PD
        RET
BITON   [ 7 ]   GR1,GR4       1のビットの個数をカウントアップ
        JUMP    CHECK
ODDBIT  OR      GR3,=#[ 8 ]   第8ビット目に1を立てる
        JUMP    SET
D       DC      #00FE
PD      DS      1
        END
```

## 10.3 データのパック・アンパック

ビット操作をうまく使うとデータをとてもコンパクトに扱うことができます。本節では、データを1語の中にパックして扱う方法とそれを取り出す方法について説明します。

**例題10-2**

時刻データ(午前のみ)が次のように1語の中に格納されています。

| ← 時刻 → | | | | ← 分 → | | | | | | ← 秒 → | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | $9A56_{(16)}$ |

ビット番号0〜5の6ビットに秒が、ビット番号6〜11の6ビットに分が、ビット番号12〜15ビットの4ビットに時刻が格納されています。たとえば、上記のデータは、9(=$1001_{(2)}$)時41(=$101001_{(2)}$)分22(=$10110_{(2)}$)秒を示しています。このように時刻が1語の中にパックされたデータが領域TIMEに格納されています。これを取り出し、時刻をHOUR、分をMINUTE、秒をSECONDに取り出すプログラム(UNPKTIME.casl)を作ってください。

このように特定の連続するビットに格納されたデータを取り出すには、AND命令で特定のビットを抽出して、シフト命令でずらすという方法が考えられます。上記のデータ$1001101001010110_{(2)}$がGR1にロードされているとしましょう。このデータに対して直接ANDやORなどのビット操作を行うと、もとのデータが破壊されてしまうので、「LD GR2,GR1」としてGR2に複写し、GR2に対してビット操作を行います。

さて、GR2から秒を取り出すには、GR2の値と$0000000000111111_{(2)}$=$003F_{(16)}$とANDを取ります(AND GR2,=#003F)。

```
     1001101001010110          9A56(16)
AND  0000000000111111      AND 003F(16)
     ----------------      ------------
     0000000000010110          0016(16)
```

すると、上記のように秒の部分だけが$010110_{(2)}$、すなわち$22_{(10)}$として抽出されます。次に、再びGR1をGR2に複写(ロード)して、今度は分を取り出します。分を取り出すには、GR2の値と$0000111111000000_{(2)}$=$0FC0_{(16)}$とANDを取ります。この一連の操作は次のようになります。

```
        LD      GR2,GR1        GR1の値をGR2に複写
        AND     GR2,=#0FC0     分の部分を抽出
```

```
     1001101001010110          9A56
AND  0000111111000000      AND 0FC0
     ----------------      --------
     0000101001000000          0A40
```

すると上記のように分の部分だけが抜き取られます。しかし、この値は$0A40_{(16)}$＝2624であり分の

数値として扱えないので、6ビット右にシフトします。この場合、算術シフトでも論理シフトでも違いはありませんが、時刻を求める時には論理シフトでないとだめ(理由は後述)なので、論理シフトに統一することにしましょう。

```
0000101001000000
      ↓(右に6ビット論理シフト)
0000000000101001
```

この値(101001(2))は、41ですから、ちょうど分が抜き出されました。

次にGR2を再度GR1と同じ値にして、時刻を取り出します。時刻を取り出すには、GR2の値と1111000000000000(2)=F000(16)とANDを取ります。

```
    1001101001010110            9A56
AND 1111000000000000       AND  F000
    ----------------      ----------
    1001000000000000            9000
```

そして、今度はこの値を12ビット右にシフトすればよいのです。しかし、このときに算術シフトを使ってはいけません。算術シフトでは、第15ビット(最上位ビット)を符号ビットとして扱いますので、もし、算術シフトで12ビット右にシフトすると、次のようになってしまいます。

```
1001000000000000
       ↓(右に12ビット算術シフト)
1111111111111001
```

この値は符号付きの場合−7、符号無しの場合65529として扱われ、もとの9という数値とは全く異なる値になっています。これでは、時刻を抜き出したことにはなりません。そこで、符号ビットを無視する論理シフトを使います。

```
1001000000000000
       ↓(右に12ビット論理シフト)
0000000000001001
```

これで、9(=1001)が取り出されました。

このプログラムを以下に示します。

```
UNPKTIME      START
              LD    GR1, A          GR1←A
; 秒を取り出す
              LD    GR2, GR1        GR2←GR1
              AND   GR2, =#003F     GR2の下位6ビットを抽出
              ST    GR2, SECOND     GR2→SECOND(秒抽出)
; 分を取り出す
              LD    GR2, GR1        GR2←GR1
              AND   GR2, =#0FC0     GR2のビット番号6〜11を抽出
              SRL   GR2, 6          シフトして桁合わせを行う
              ST    GR2, MINUTE     GR2→MINUTE(分抽出)
; 時刻を取り出す
              LD    GR2, GR1        GR2←GR1
              AND   GR2, =#F000     GR2のビット番号12〜15を抽出
              SRL   GR2, 12         シフトして桁合わせを行う
              ST    GR2, HOUR       GR2→HOUR(時刻抽出)
              RET
A             DC    #9A56
HOUR          DS    1
MINUTE        DS    1
SECOND        DS    1
              END
```

## 演習問題10-3

(1) GR4から第4〜7ビット目を数値として抽出する命令を書いてください。

(2) GR4から第5〜12ビット目を数値として抽出する命令を書いてください。

(3) GR4から第1〜14ビット目を数値として抽出する命令を書いてください。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GR4 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

|← (1)抽出 →| (ビット7〜4)

|← (2)抽出 →| (ビット12〜5)

|← (3)抽出 →| (ビット14〜1)

(4) 4桁の10進数が次のように格納されています。全体を4ビットづつ区切って、その中に10進数の第1桁から第4桁の数値が対応する2進数に変換されています。たとえば、9602という10進数ならば次のように格納されています。なお、このように表現されたデータを**パック10進数**と呼びます(＊1)。

| 4桁目 | | | | 3桁目 | | | | 2桁目 | | | | 1桁目 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 9 | | | | 6 | | | | 0 | | | | 2 | | | |

次のプログラム(DECBIN)は、パック10進数、たとえば、$9602_{(16)}$を10進数の9602(=$2582_{(16)}$=$0010010110000010_{(2)}$)に変換するプログラムです。空欄に適当な字句を入れてください。

(＊1) 実際のパック10進数はデータの最後に4ビットの符号表現(通常、＋なら1100、－なら1101)が入ります。

考え方：各桁の数値を取り出して、$9 \times 10^3 + 6 \times 10^2 + 0 \times 10^1 + 2$を計算しています。最初は1桁目を計算しますが、これは1を2回加えます。次に2桁目を計算しますが、これは10を0回加えます。次に3桁目を計算しますが、これは100を6回加えます。次に4桁目を計算しますが、これは1000を9回加えます。

```
DECBIN  START
        LD      GR1,PDEC    GR1←PDEC
        [ 1 ]   GR2,GR2     GR2←0, ここに10進数を求める
        LAD     GR3,1       1(定数), 計算用
; 1桁目を取り出す
        LAD     GR4,1       1=1桁目の重み
        LD      GR5,[ 2 ]       元データをGR5に待避してビット操作する
        AND     GR5,=#000F  下1桁目を取り出す
LOOP1   [ 3 ]   DIGIT2      ゼロならば2桁めの処理へ(GR5回ループを繰り返す)
        ADDA    GR2,GR4     この桁の重み(1)を加える
        SUBA    [ 4 ],GR3   カウントダウン
        JUMP    LOOP1       LOOP1へ
; 2桁目を取り出す
DIGIT2  LAD     GR4,10      10=2桁目の重み
        LD      GR5,[ 2 ]       元データをGR5に待避してビット操作する
        AND     GR5,=#00F0  下2桁目を取り出す
        [ 5 ]   GR5,4       その桁を最下位へずらす
LOOP2   JZE     DIGIT3      ゼロならば3桁めの処理へ(GR5回ループを繰り返す)
        ADDA    GR2,GR4     この桁の重み(10)を加える
        SUBA    GR5,GR3     カウントダウン
        JUMP    LOOP2       LOOP2へ
; 3桁目を取り出す
DIGIT3  LAD     GR4,100     100=3桁目の重み
        LD      GR5,[ 2 ]       元データをGR5に待避してビット操作する
        AND     GR5,=#[ 6 ]     下3桁目を取り出す
        SRL     GR5,8       その桁を最下位へずらす
LOOP3   JZE     DIGIT4      ゼロならば4桁めの処理へ(GR5回ループを繰り返す)
        ADDA    GR2,GR4     この桁の重み(100)を加える
        [ 7 ]   GR5,GR3     カウントダウン
        JUMP    LOOP3       LOOP3へ
; 4桁目を取り出す
DIGIT4  LAD     GR4,1000    1000=3桁目の重み
        LD      GR5,[ 2 ]       元データをGR5に待避してビット操作する
        AND     GR5,=#F000  下4桁目を取り出す
        SRL     GR5,[ 8 ]       その桁を最下位へずらす
                    LOOP4 JZE          [ 9 ]       ゼロならば終了処理へ
(GR5回ループを繰り返す)
        ADDA    GR2,GR4     この桁の重み(1000)を加える
        SUBA    GR5,GR3     カウントダウン
        JUMP    LOOP4       LOOP4へ
; 答えを格納する
FINAL   ST      GR2,ANS
        RET
PDEC    DC      #9602
ANS     DS      1
        END
```

## 10.4 データのビットの左右反転

> **例題10-3**
>
> たとえば、A=0101100101011010(2) (=595A(16))というデータのビットを左右反転して0101101010011010(2) (=5A9A(16))というデータをREVへ格納するプログラム(REVERSE.casl)を作成してください。

これは、次のようにすればよいでしょう。

(1) GR1←A
(2) GR2←0(ここに反転されたデータを格納する)
(3) GR3←1000000000000000(2)をセットする
(4) GR1を1ビット右論理シフトして最下位ビットを押し出す。
(5) もし、それが1ならばGR3をGR2に加える
(6) GR3を1ビット右論理シフトする
(7) その結果、もしオーバーフローしたらGR3の1つしかない1のビットが押し出されたということなので(9)へジャンプして終了
(8) (4)へジャンプする
(9) GR2→REV

```
REVERSE  START
         LD    GR1,A       GR1←A
         XOR   GR2,GR2     GR2←0, ここに反転されたデータを格納する
         LAD   GR3,#8000   GR3←1000000000000000をセットする
LOOP     SRL   GR1,1       GR1を1ビット右論理シフトして最下位ビットを押し出す
         JOV   SETDATA     もし、それが1ならばGR3をGR2に加える
CONT     SRL   GR3,1       GR3を1ビット右論理シフトする
         JOV   FINISH      オーバーフローしたら終了するために(9)へジャンプ
         JUMP  LOOP        ループ継続
SETDATA  ADDA  GR2,GR3     反転したビットをセット
         JUMP  CONT        主処理へ戻る
FINISH   ST    GR2,REV     GR2→REV
         RET               ;プログラムの終了
A        DC    #595A
REV      DS    1
         END
```

なお、この例では、16回ループを回すのに、ビット操作を使っていることに注目してください。最初、上記の(3)でGR3の最上位ビットだけを1にして、1回ループを回ったら1ビット論理右シフトします。そして、もし、その1が押し出されてオーバーフローしたら、それは16ビットシフトされたことを意味しますから、その時点でループを終了すれば良いのです。同様に、最初、1をレジスタにセットしておき、ループを1回ったら1ビット論理左シフトします。そして、もし、その1が押し出されてオーバーフローしたら、それは16ビットシフトされたことを意味しますから、その時点でループを終

了するという方法もあります。この例は、オーバーフローを繰り返しの条件としている例といえます。

## 演習問題10-4

次のプログラム(BITSET.casl)は、領域Aの16ビットのデータの1のビット数を数えて、その数をNとします。次に、第0ビット目から第N−1ビット目だけが1のデータ(16ビット)を作り、Bに格納するプログラムです。

たとえば、次の例では、Aにある1のビット数は5です。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

そこで、Bの内容は次のようになります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

←― 5つ ―→

なお、このプログラムの作り方として、ビットの数を数え終わってNを求めてからBのデータを作るという方法も考えられますが、以下のプログラムではビットの数を数えながら，同時にデータを作っています。

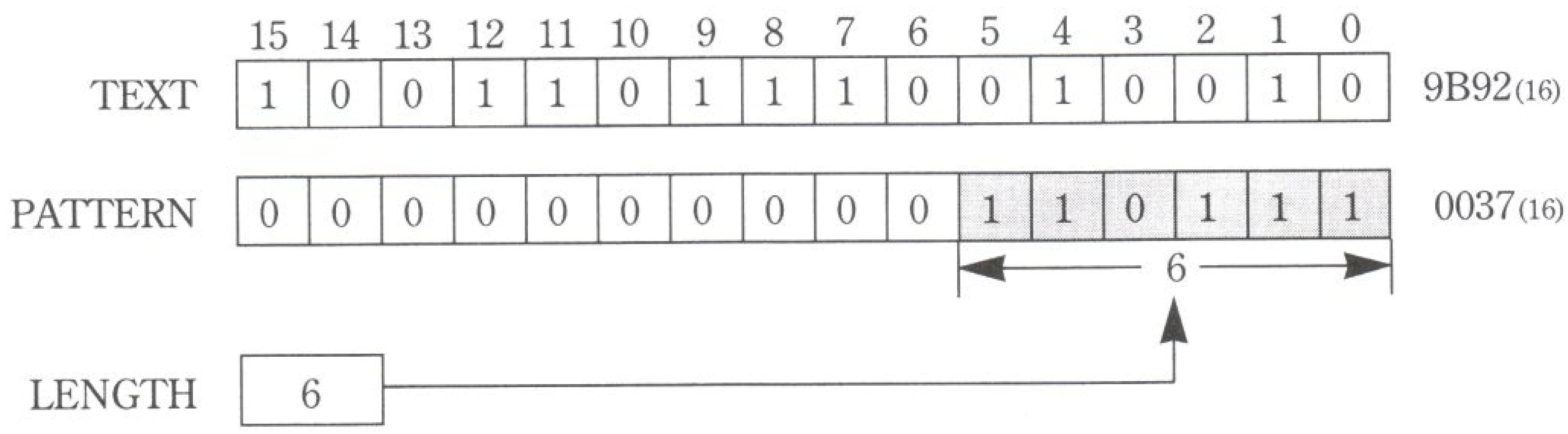
空欄に適当な字句を入れてください。

```
BITSET   START
         LD     GR1,A
         XOR    GR2,GR2     ここにデータを作る
         LAD    GR3,#0001   GR3←0000000000000001、マスクデータ
         LAD    GR4,#0001   GR4←0000000000000001、カウント用のデータ
LOOP     SRL    GR1,1       GR1(Aの値)を1ビット右論理シフト
         [ 1 ]  SETDATA     Aからデータが押し出されたらデータをセット
CONT     SLL    GR4,1       カウントデータを左論理シフト
         JOV    FINISH      カウントデータから1のビットが出たら終了
         JUMP   LOOP
FINISH   ST     GR2,B
         RET
SETDATA  OR     GR2,GR3     ベースデータのビットをGR2にセット
         [      2      ]    マスクデータを1つ左にずらす
         JUMP   CONT
A        DC     #9602
B        DS     1
         END
```

## 10.5 ビットのパターン検索

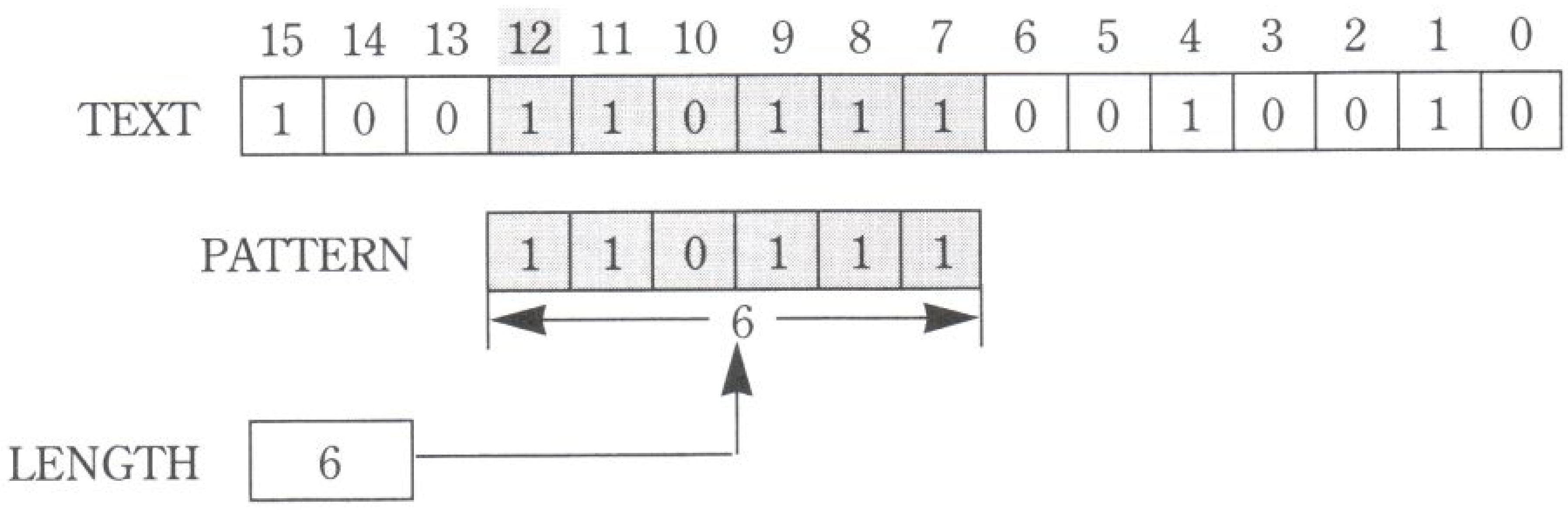
**例題10-4**

次のような3つのデータがあるとします。

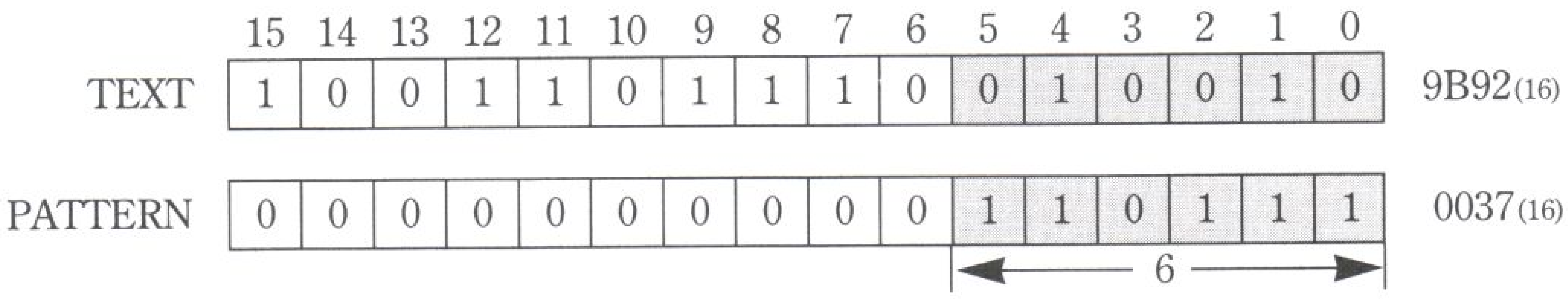



TEXT中のデータの中から、PATTERNに格納されたデータに一致する部分を見つけてそのビット番号を返すプログラム(BITFOUND.casl)を作成してください。ただし、PATTERN中のチェック対象になるビット数は、LENGTHに格納されています。

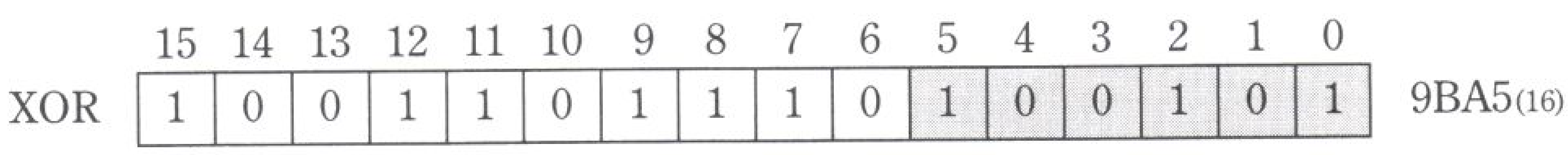
例題中に示すようにPATTERNの長さが6ならば、まず最初にTEXTの第0ビット～第5ビットが検索対象となります。そしてTEXTの検索部分を次々にずらしながら、この例では、次の部分でTEXT中の第7ビットから第12ビットにPATTERNと一致する部分が見つかります。




この場合、見つかった位置の12をPOSに格納します。この位置を発見する方法は単純です。PATTERN中の検索部分を右側からずらしてチェックしていきます。最初は、もっとも右側でチェックします。まず、この2つ(TEXTとPATTERN)のXORをとります。




TEXTとPATTERNのXORを取ると、次のようになります。




もし、TEXTとPATTERNの検索部分が同じならそこにはゼロがセットされます。検索部分(上図の網掛け部分)のみを取り出すために、右側からのビットの長さが6のマスクデータを用意します。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| マスク | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 003F(16) |

そのマスクデータ(003F(16))とXORの結果(9BA5(16))のANDをとります。

```
    1001101110100101        9BA5
AND 0000000000111111    AND 003F
    0000000000100101        0025
```

その結果は、次のようになります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AND | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0025(16) |

TEXTとPATTERNの検索部分が一致していれば，この値はゼロのはずです。しかし、この場合は一致していないのでゼロではありません。そこで、PATTERNとマスクデータをそれぞれ1ビット左に論理シフトして同じ処理を行います(TEXTはシフトしません)。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TEXT | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 9B92(16) | |
| PATTERN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 006E(16) | ←左に1ビットシフト |

この2つのXORを取ると次のようになります。

```
    1001101110010010        9B92
XOR 0000000001101110    XOR 006E
    1001101111111100        9BFC
```

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| XOR | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 9BFC(16) |

さらに、前述のようにマスクデータも左に1ビット論理シフトします。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| マスク | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 007E(16) | ←左に1ビットシフト |

XORの結果とマスクのANDを取ります。

```
    1001101111111100        9BFC
AND 0000000001111110    AND 007E
    0000000001111100        007C
```

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AND | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 007C(16) |

この値も結局ゼロではありません。しかし、次々に1ビットづつ左にシフトしていくと、やがて次

のようになります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TEXT | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 9B92(16) | |
| PATTERN | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1B80(16) | ←左に最初から数えて7ビットシフト |

(ビット12〜7の範囲: 6)

この2つのXORを取ると次のようになります。

```
    1001101110010010          9B92
XOR 0001101110000000      XOR 1B80
    1000000000010010          8012
```

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| XOR | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 8012(16) |

この値と7ビット左にシフトしたマスクの論理積をとり、PATTERNの部分を抽出します。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| マスク | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1F80(16) | ←最初から数えて左に7ビットシフト |

```
    1000000000010010          8012
AND 0001111110000000      AND 1F80
    0000000000000000          0000
```

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AND | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000(16) |

今度は, ANDを取ったビットが全部ゼロになり、ビット番号12の場所でビット列が一致していることがわかります。以上の方針で作成したプログラムを以下に示します。

```
BITFOUND  START
          LD    GR0,TEXT      GR0←TEXTの値
          LD    GR1,PATTERN   GR1←PATTERNの値
          LD    GR2,LENGTH    GR2←LENGTHの値
; 長さLENGTHのマスクを作る
          XOR   GR3,GR3       マスクデータをゼロにする
          LAD   GR4,1         マスクビットデータ初期値
          LAD   GR5,1         計算用定数
          LD    GR6,GR2       長さをロード
MKMASK    JZE   SETFIN        長さがゼロになったら次へ
          OR    GR3,GR4       マスクに1を設定
          SLL   GR4,1         マスクビットデータをずらす
          SUBA  GR6,GR5
          JUMP  MKMASK
; 検索する
SETFIN    SUBA  GR2,GR5       GR2←N-1,発見される位置
FLOOP     LD    GR6,GR0       GR6:検索されるデータ
          XOR   GR6,GR1       検索されるデータとPATTERNのXOR → 一致すればそこだけゼロ
          AND   GR6,GR3       その部分だけを取り出す
          JZE   FINISH        それがゼロなら発見
          SLL   GR1,1         パターンを一つずらす
          SLL   GR3,1         マスクデータを一つずらす
          JOV   NOTFOUND      マスクデータから1ビットでも押し出されたら発見できずに終わり
          ADDA  GR2,GR5       発見される位置を左へずらす
NOTFOUND  LAD   GR2,-1        発見できない
          JUMP  FLOOP
FINISH    ST    GR2,POS
          RET
TEXT      DC    #9B92
PATTERN   DC    #0037
LENGTH    DC    6
POS       DS    1
          END
```

## 演習問題10-5

領域Aの中にある連続した1のビット列の長さを右から左に向かって調べ、領域Nで与えられた数(N>0)と同じ長さのビット列があるかどうか判定し、一致するものがあれば、そのビット番号をPに格納するプログラムを以下に示します。一致するものがない場合には、Pには−1を入れてください。たとえば、Aの内容が次のようであり、Nの値が5であれば、Pの値は10になります。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | $E7CA_{(16)}$ |

(ビット10〜6：←5→)

このAに対して、Nの値が4のときには、連続する4つの1がないと考え、Pの値は－1になります。Nの値が3のときはPの値は15、Nの値が1のときはPの値は1になります。この場合、Pの値として3も候補になりますが、右から左に向かって調べていくので、ビット番号の低い方を選びます。

(1) 空欄に適当な字句を入れてください。

```
CBITFND  START
         LD      GR0,A
         LD      GR1,N
         XOR     GR2,GR2    現在の連続した1の個数、最初は0
         LAD     GR3,1      GR3←1(タイミング処理用)
         LAD     GR4,1      定数、カウントアップ用
         LAD     GR5,[ 1 ]        位置、最初は1つ手前にしておく
LOOP     [      2      ]    データからビットを取り出す
         JOV     BIT1       1が抽出された
         CPA     GR2,GR1    連続した1の長さがN(=GR1)と等しいか
         [ 3 ]   FOUND      等しければ
         XOR     GR2,GR2    連続したビットの個数を0に戻す
CONTI    SLL     GR3,1      カウンタを1つ左シフト
         JOV     FINISH     GR3の唯一のビットが飛び出たら終了
         [      4      ]    発見位置をカウントアップ
         JUMP    LOOP       ループを繰り返す
BIT1     ADDA    GR2,GR4    連続する長さをカウントアップ
         JUMP    CONTI      本流に戻る
FINISH   ADDA    GR5,GR4    もし、ここにあるとしたら位置を+1しておく必要がある
         [ 5 ]   GR2,GR1    ここでもう一度チェック
         JZE     FOUND      一致したら、長さはGR5に格納されている
         LAD     GR5,[ 6 ]        結局見つからなかった
FOUND    ST      GR5,P
         RET
A        DC      #E7CA
N        DC      5
P        DS      1
         END
```

ヒント：たとえば、次のように左側に1のビットが連続してある場合に、終了の印となる0がありません。FINISH時の処理では、この処理を行っています。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

←6→（ビット15〜10）

(2) このプログラム(CBITFND.casl)の境界条件となるようなデータを考えてください。

## 10.6 効率の良い乗算

**例題10-5**

乗算のプログラムを2進演算で考えましょう。
たとえば、$100101_{(2)} \times 1011_{(2)}$=37×11=407=$110010111_{(2)}$の計算は次のようになります。

```
     100101
  ×    1011
-----------
     100101
    1001010
   00000000
+ 100101000
-----------
  110010111
```

参考

```
(a)  100101     (b)  100101     (c)    100101     (d)    100101
   ×      1        ×     10        ×      000        ×     1000
   ---------       ---------       -----------       -----------
     100101         1001010         00000000         100101000
```

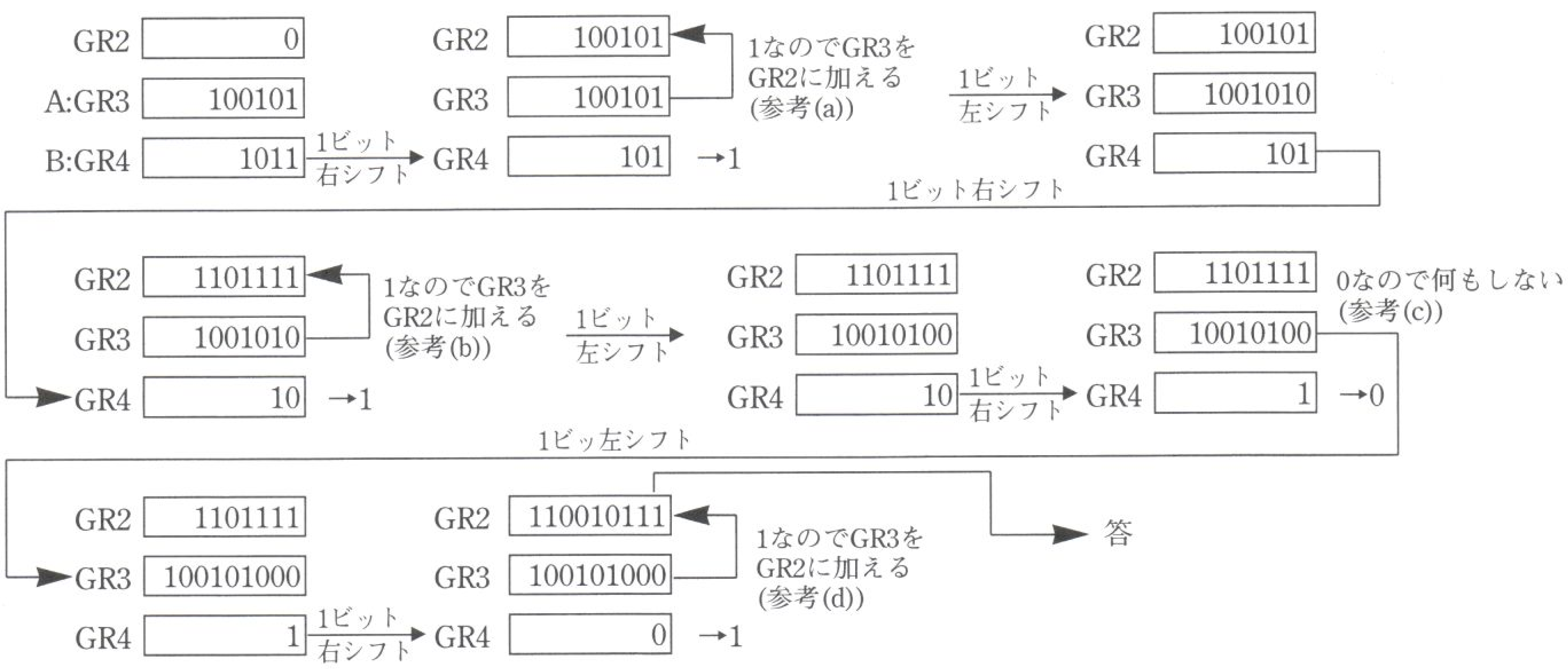
この計算メカニズムに基づいてA(≧0)×B(≧0)を計算するプログラム(BMULT.casl)を作成してください。ただし、計算途中でオーバーフローした場合、OVFに1をセットし、計算は続行させてください。また、オーバーフローしない場合には、OVFには0をセットしてください。

まず、A=$100101_{(2)}$, B=$1011_{(2)}$としましょう。

流れは次のようになるでしょう。

(1) GR1←0(オーバーフローフラグOVF)
(2) GR2←0(解)
(3) GR3←A
(4) GR4←B
(5) GR5←16(カウンタ、16ビット分操作する)
(6) GR6←1(計算用)
(7) GR4を1ビット右にシフトして、1ならばGR2にGR3を加える
(8) オーバーフローしたらGR1を1にする
(9) GR3を1ビット左にシフトする
(10) GR5をカウントダウン
(11) ゼロでなければ(7)へ

この方法では、どんなデータであってもループ回数は16回なので効率が良いといえます。なお、この流れを図にて説明します。




```
BMULT     START
          XOR    GR1,GR1     GR1←0, オーバーフローフラグ(0:いない,1:いる)
          XOR    GR2,GR2     GR2←0, 解を格納するレジスタ
          LD     GR3,A       GR3←A
          JZE    FINISH      もしAがゼロなら計算しない
          LD     GR4,B       GR4←B
          JZE    FINISH      もしBがゼロなら計算しない
          LAD    GR5,16      GR5←16, 16回ループする(GR5はカウンタ)
          LAD    GR6,1       GR6←1, 計算用定数
LOOP      SRA    GR4,1       GR4(B)のビットを押し出す
          JOV    ADDVALUE    押し出された最下位ビットが1なら加算処理へ
CONT      SLA    GR3,1       Aを1ビット左へシフト
          SUBA   GR5,GR6     カウントダウン
          JNZ    LOOP        ゼロでなければLOOPへ
FINISH    ST     GR1,OVF     GR1→OVF
          ST     GR2,C       GR2→C
          RET                ;プログラムの終了
ADDVALUE  ADDA   GR2,GR3     解(GR2)に重み付けされたA(GR3)を加える
          JOV    SETFLAG     もし、オーバーフローしたならフラグをセット
          JUMP   CONT        主処理へ戻る
SETFLAG   LAD    GR1,1       オーバーフローフラグをセット
          JUMP   CONT        主処理へ戻る
A         DC     37
B         DC     11
C         DS     1
OVF       DS     1
          END
```

# 演習問題10-6

符号を考慮した乗算(C←A×B)のプログラムをBMULTをベースにして開発したものが以下のプログラムです。

符号の処理は次のように前処理・後処理で行っています。

**表 10-2 符号の反転処理**

| | | B<br>＋ | B<br>－ |
|---|---|---|---|
| A | ＋ | なにもしない | 前処理：Bの符号反転<br>後処理：Cの符号反転 |
| A | － | 前処理:Aの符号反転<br>後処理:Cの符号反転 | 前処理：A,Bの符号反転 |

後処理は、レジスタGR0の値が0なら「何もしない」、1なら「Cの符号を反転する」処理を行います。次のプログラムの空欄に適当な字句を入れてください。

```
BMULTS    START
          XOR     GR0,GR0      後処理フラグ
          LAD     GR7,#FFFF    マスク
          LAD     GR6,1        1,定数
; Aのロードと符号判定
          LD      GR3,A        Aのロード
          JPL     BPROC        正なら次へスキップ
          JZE     [  1  ]      ゼロなら計算しなくてもゼロ
          XOR     GR3,GR7      A(GR3)の符号
          ADDA    GR3,GR6      を反転
          [ 2 ]   GR0,GR6      GR0レジスタを反転
; Bのロードと符号判定
BPROC     LD      GR4,B        Bのロード
          JPL     CAL          正なら何もせずに計算へ
          JZE     [  1  ]      ゼロなら計算しなくてもゼロ
          XOR     GR3,GR7      B(GR4)の符号
          ADDA    GR3,GR6      を反転
          [ 2 ]   GR0,GR6      GR0レジスタを反転
```

```
; 本計算
CAL         XOR    GR1,GR1     GR1←0, オーバーフローフラグして(0:いない,1:いる)
            XOR    GR2,GR2     GR2←0, 解を格納するレジスタ
            LAD    GR5,16      GR5←16, 16回ループする(GR5はカウンタ)
LOOP        SRA    GR4,1       GR4(B)のビットを押し出す
             JOV   ADDVALUE    押し出された最下位ビットが1なら加算処理へ
CONT        SLA    GR3,1       Aを1ビット左へシフト
            SUBA   GR5,GR6     カウントダウン
            JNZ    LOOP        ゼロでなければLOOPへ
FINISH      ST     GR1,OVF     GR1→OVF
; GR0の値が1なら符号反転
            CPA    GR0,GR6
            [ 3 ]  MINUSC
SETC        ST     GR2,C       GR2→C
            RET                ; プログラムの終了
ADDVALUE    ADDA   GR2,GR3     解(GR2)に重み付けされたA(GR3)を加える
            JOV    SETFLAG     もし、オーバーフローしたならフラグをセット
            JUMP   CONT        主処理へ戻る
SETFLAG     LAD    GR1,1       オーバーフローフラグをセット
            JUMP   CONT        主処理へ戻る
MINUSC      XOR    GR2,GR7
            ADDA   GR2,GR6
            JUMP   [ 4 ]
A           DC     37
B           DC     -11
C           DS     1
OVF         DS     1
            END
```

# 総合問題10

**[演習10-1]** 次の各処理の方針を述べてください。

(1) 特定領域(ある語のビット番号3〜12など)の1のビットの数を数える。
(2) ループの中でGR2の値を、1,0,1,0,1,0,…と交互に分岐条件を使わずに動かす。
(3) 特定領域のビットを小さな語とみなして、その部分の2の補数を取る。他のビット(以下の図の網のない部分)は0にする。

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

(ビット11〜6:)この中を1語とみなして2の補数

(4) 2つの語の同一ビットの連続する部分、たとえば、(11…1あるいは00…0)が12ビット以上であることを判別する。
(5) 3つの語のビットがすべて一致することを判定する。
(6) 1語の中のビットを4ビットごとに区切って、次のように交換する。
ABCD(16)ならBDCA(16)
(7) ある語の第0ビットから第3ビットを、第4ビットから第7ビット、第8ビットから第11ビット、第12ビットから第15ビットへ複写する。たとえば、ABCD(16)ならDDDD(16)とする。

**[演習10-2]** 次の計算を実際に行う(すべてループで実現)プログラムを作成してください。

(1) S＝1 − 2 ＋ 3 − 4 ＋ 5 − 6 ＋…… ± N (最後の演算は＋か−か分からないので±にした)
(2) S＝1 − 2 ＋ 4 − 8 ＋ 16 − 32 ＋ 64 − 128 ＋ 256

**[演習10-3]** 次の表は、第t回目のループの処理の内容を表しています。
たとえば、2回目のループではAを行って5回目のループではBを行うことを示してします。

| ループ回数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 処理内容 | A | A | B | B | B | A | B | A | A | A | B | B | A | B | A | A |

処理内容としてAが来るかBが来るかは自由に決められる(変更できる)ようにしたいのです。このループ処理を簡潔に書くにはどうしたらよいでしょうか。

[ヒント] 処理内容がAとBの2つしかないことに注目する。

**[演習10-4]** 例題10-1において、領域Aの1のビット数が少ない場合(あるいはビット番号の少ない方へ集中している場合)、Aの値を格納してループの中でシフトしているレジスタGR1の値がゼロになったのであれば、それ以上集計する必要がないことがわかります。そこで、この処理を組込んでプログラムBCOUNTを効率化したプログラムBCOUNTEを作成してください。

# 第11章 テーブル操作

連続した領域のことを一般に**テーブル**あるいは**表**と呼びます(*1)。また、テーブルに関する処理を、**テーブル操作**あるいは**表操作**と呼びます。テーブルを使うと同じ種類のデータを大量に扱うことができます。そのため、テーブル操作は、CASL II の中でもさまざまなバリエーションのプログラミングを提供します。

テーブル操作によって行われる処理の一覧を表11-1に示します。

**表11-1 テーブル操作によって実現されるプログラム**

| 名　前 | 章節 | 処　理 | 説　明 |
|---|---|---|---|
| テーブルセット | 11.2 | ゼロ設定、連番設定 | 連続するデータ領域(テーブル)にデータを設定する |
| テーブル集計 | 11.3 | 総和 | テーブル中のデータを集計する |
| テーブルサーチ | 11.4 | データ件数　最大値・最小値 | テーブル中のデータを検索する |
| 編集処理 | 11.5 | 10進文字列変換、16進文字列変換、金額編集 | データを編集してテーブルに文字として格納 |
| 文字列処理 | | 検索 挿入 削除 | テーブル中の文字列を処理 |
| 整列 | | 並べ替え(降順・昇順) | テーブル中のデータを並び替える |
| 併合 | | 給与計算、マスターファイル更新 | 2つのテーブルを処理する |

本章では、まず、11.1節でテーブルの概要について説明し、11.2節ではテーブル処理の基礎を「データを設定する」という題材(テーブルセット)を通して説明していきたいと思います。11.3節〜11.5節では、表11.1に示す内容を説明します。

また、今までは、プログラムと言っても、単にレジスタの値を設定したり計算したりするプログラムがほとんどでしたが、11.5節で学習する編集処理では、「数値データ」を「文字データ」へ変換する方法を学習します。この処理を用いると、レジスタや主記憶装置の値を目で見て分かるように表示させることができます。また、テーブルに格納する値として単なる数値だけでなくアドレスを保持させることもできます。たとえば、実行すべき命令のアドレスをテーブルに保持し、そのテーブル中のアドレスにジャンプするプログラムを作ることがありますが、このような使い方を**ジャンプテーブル**と呼びます。ジャンプテーブルについては、6.6(3)などで説明しています。

(*1)高水準言語では、配列と呼ぶこともあります。

# 11.1 テーブルとは

連続したデータを一般に、**テーブル**(table)あるいは表と呼びます。たとえば、次に示すデータは、5人分の男性の身長のデータです。

**表11-2 男性の身長**

| 168 |
|---|
| 176 |
| 173 |
| 167 |
| 170 |

このデータは、CASL IIでは、次のように記述します。

```
HEIGHT  DC    168
        DC    176
        DC    173
        DC    167
        DC    170
```

あるいは、横1行にして、次のようにも記述できます。

```
HEIGHT  DC    168,176,173,167,170
```

なお、このとき、各数値間(コンマも含めて)には空白を書かないように注意してください。空白を書くと、それ以降がコメントとして扱われてしまいます。このDC命令によって、主記憶装置内に次のようにデータが確保されます。

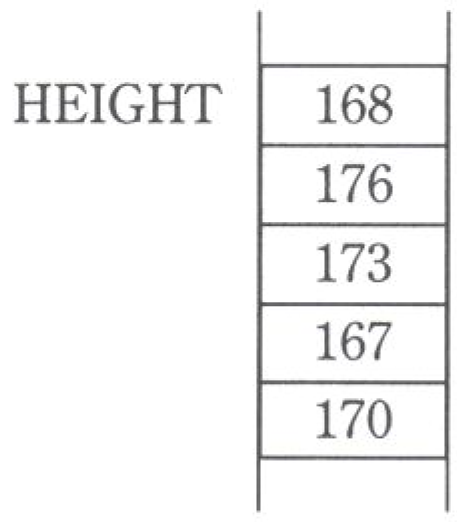



**図11-1 テーブルの主記憶装置への配置**

このテーブル内のデータを操作することが、テーブル操作です。たとえば、この中から一番高い身長を検索したり、平均身長を求めたり、身長の高い順に並び替えたりする操作などがあります。

なお、上記のようにデータの設定をしないでテーブルを用意する場合には、DS命令を使います。たとえば、100個の語を持つテーブルを用意するには、次のようにします。

```
TABLE   DS    100
```

この方法では、TABLE番地から100語用意されますが、初期値は設定されませんので各語にはどの

ような値が入っているかわかりません。

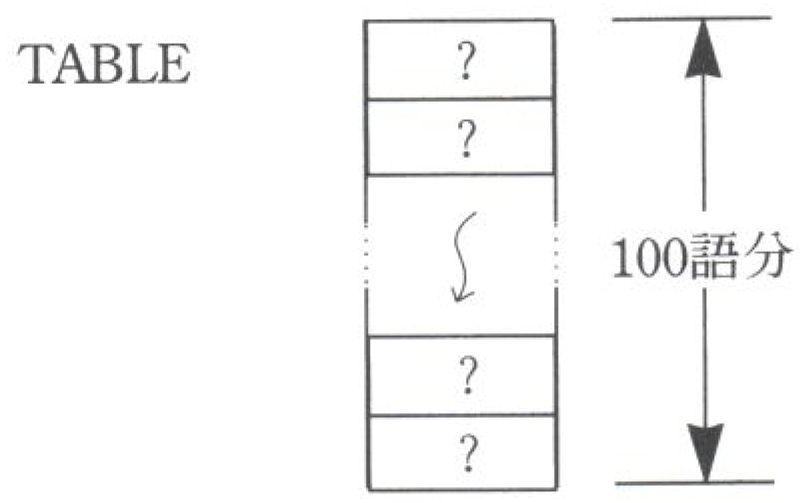



**図11-2 DS命令による配置**

テーブル操作の基本は、ループと指標レジスタの組み合わせです。この二つのテクニックを使って、テーブルの中を調べていきます。この方法は、11.2.1項「テーブルに値をセットする」で詳細に説明します。

ところで、ループについてまだ十分に学習がすんでいない人は第9章の「繰り返し型のプログラミング」をよく読んでください。また、指標レジスタについてよく分かっていない人は5.3節「指標レジスタの利用」をよく読んでください。この二つの理解無しに本章は学習できません。

## 11.2 テーブルセット

表(テーブル)に値をセットすることを**テーブルセット**と呼びます。これには、テーブルにゼロをセットする、テーブルに値をセットする、テーブルにある計算を行った結果をセットする、などさまざまなバリエーションが考えられます。いずれも、ループと指標レジスタを組み合わせることで実現できます。

それでは、次の20個分のテーブルVALTBLにデータをセットするプログラムを考えましょう。

```
VALTBL  DS    20
```

このとき、プログラムの組み方によって、テーブルの上(アドレスの小さい方)からデータをセットしていく方法とテーブルの下からデータをセットしていく方法の2つが考えられます。次の例は、値123を格納している例です（どちらも最初の2語分だけ123をセットしています)。

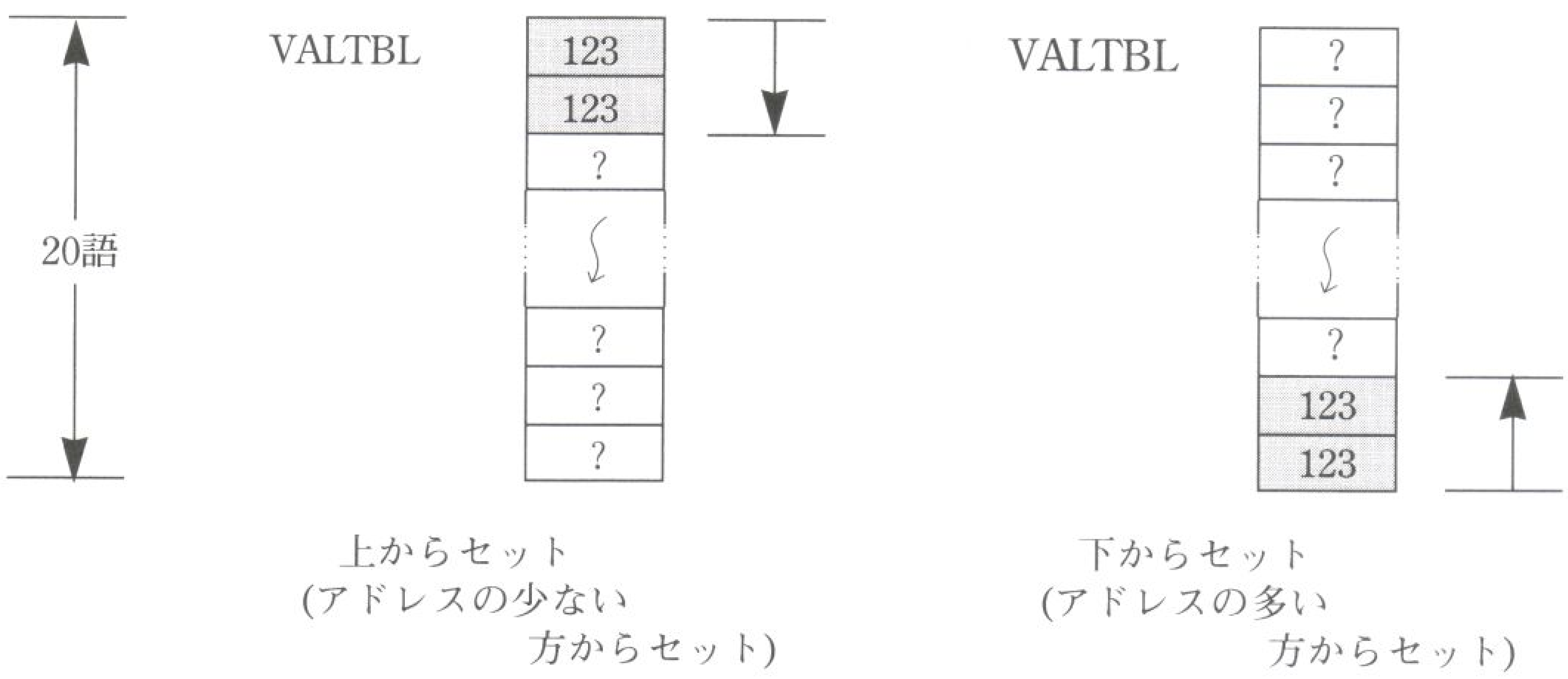



**図11-3 テーブル処理の方向**

### 11.2.1 テーブルに値を上からセットする方法：カウントアップ

> **例題 11-1**
>
> テーブル VALTBL 中の語(全部で20語)のすべてに123というデータをアドレスの少ない方からセットしてください(VALSET1.casl)。

基本的な考え方を説明しましょう。まず、テーブルにデータを上からセットしていく方法について説明します。汎用レジスタGR2に格納されている値を、領域VALTBLの全ての語に格納することを考えます。GR1を指標レジスタとして使います。GR1をループを用いて 0, 1, 2, ..., 19と動かします。そして、そのループの中で「ST GR2,VALTBL,GR1」と書けば、データ(GR2)をテーブル中のすべての語に転送(ストア)することができます。復習になりますが、この命令では「VALTBLのアドレス＋GR1の値」が実効アドレスになります。もし、VALTBLが100番地に格納されていてGR1の値が4ならば、「100＋4=104」番地が実効アドレスになります。この実効アドレスの場所にGR2の値が転送(ストア)

されます。

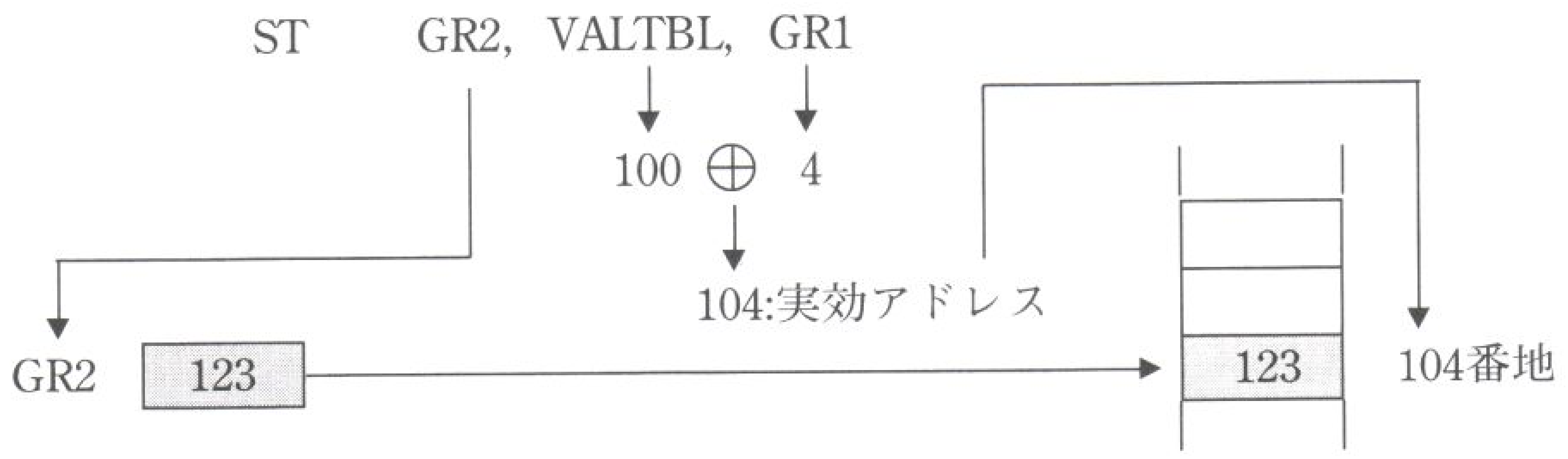



**図11-4 指標アドレスと実効アドレス**

実際の主記憶装置とレジスタGR1の関係は次の図のようになります。これで、GR1を0から19まで動かせば、テーブルに値が入っていく様子が理解できると思います。

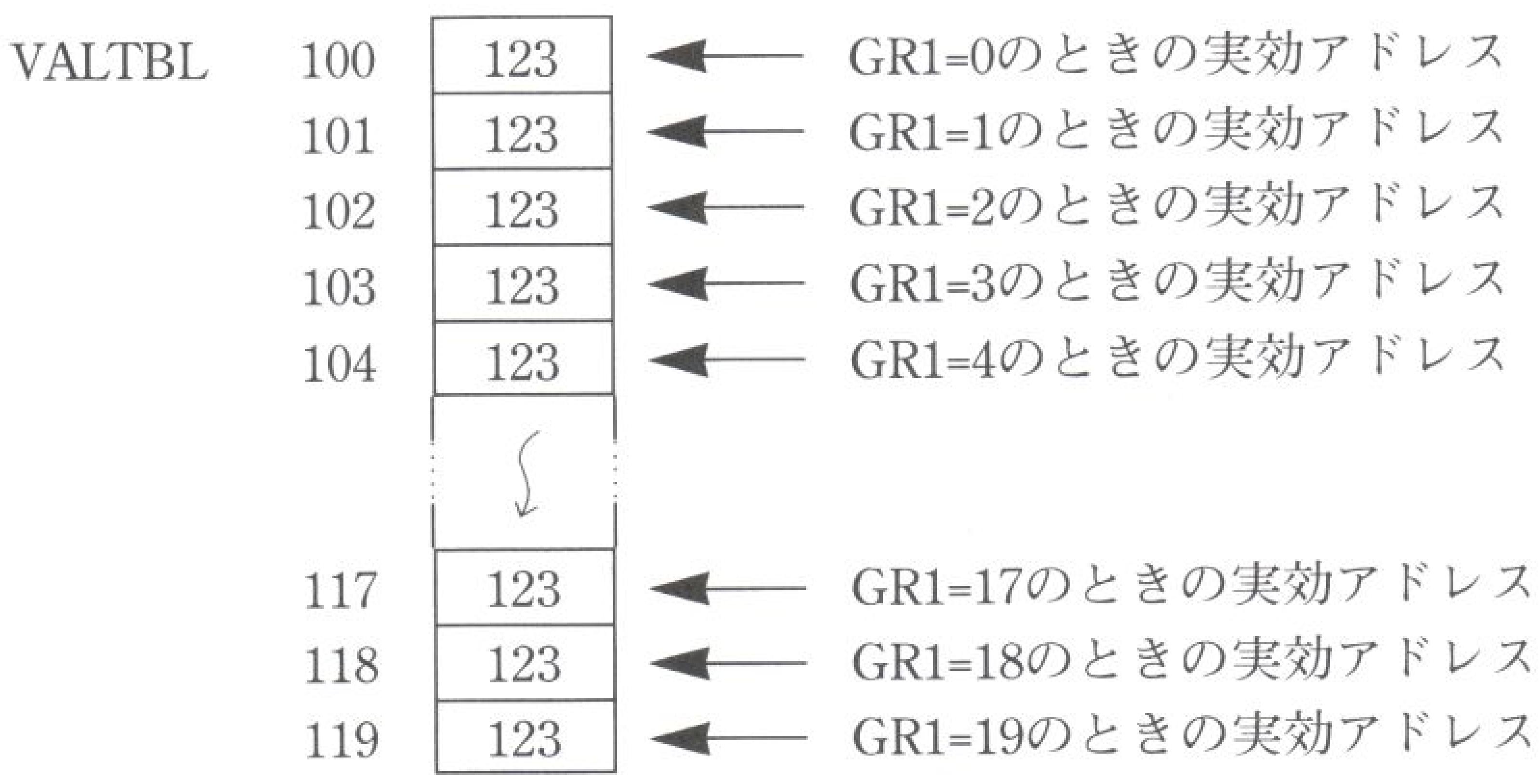



**図11-5 主記憶装置とレジスタGR1の関係**

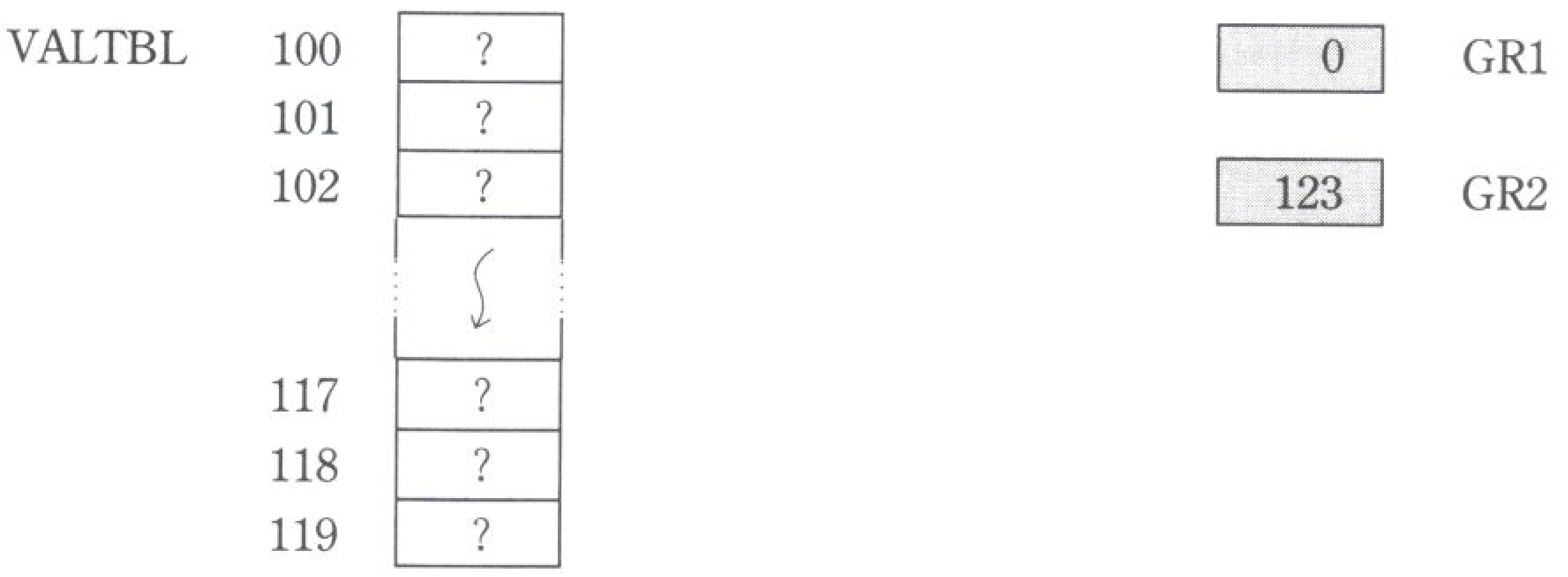
このことから、テーブルVALTBLに値123をセットするプログラムは次のようになります。

```
VALSET1  START
; 初期設定 ====================================================
         XOR     GR1,GR1          GR1:カウンタ(0,1,2,...,19と動かす)
         LAD     GR2,123          GR2:データ123
         LAD     GR3,1            GR3(=1):定数1(カウントアップ用)
         LAD     GR4,20           GR4(=20):語数=カウンタ上限−1
; テーブルセットのループ ========================================
LOOP     ST      GR2,VALTBL,GR1   GR2(=123)→VALTBLのGR1番目
         ADDA    GR1,GR3          GR1をカウントアップ
         CPA     GR1,GR4          GR1と20を比較
         JMI     LOOP             GR1<20ならLOOPへ
; 終了 ========================================================
                 RET              ;プログラムの終了
VALTBL   DS      20               20個分のデータを確保
                 END
```

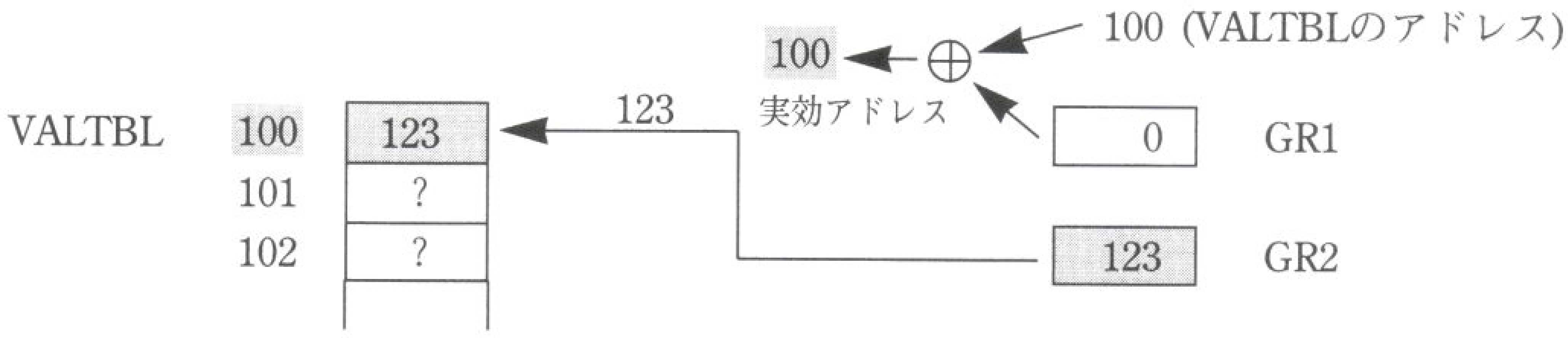
このプログラムでは、すでに説明したように、GR1の値を0, 1, 2, ..., 19とループの中で動かしながら、

「ST GR2,VALTBL,GR1」で値をセットしています。また、このループでは、ループの継続条件を「GR1<20」としているので、GR1の最終値より一つ多い値20をレジスタGR4に格納して、比較命令「CPA　GR1,GR4」で使っています。この20という数値は、このケースではテーブル VALTBL の語数と考えることもできます。

このプログラムの実行の様子を図で示しましょう。初期設定が終了した段階で、次のようになっています。



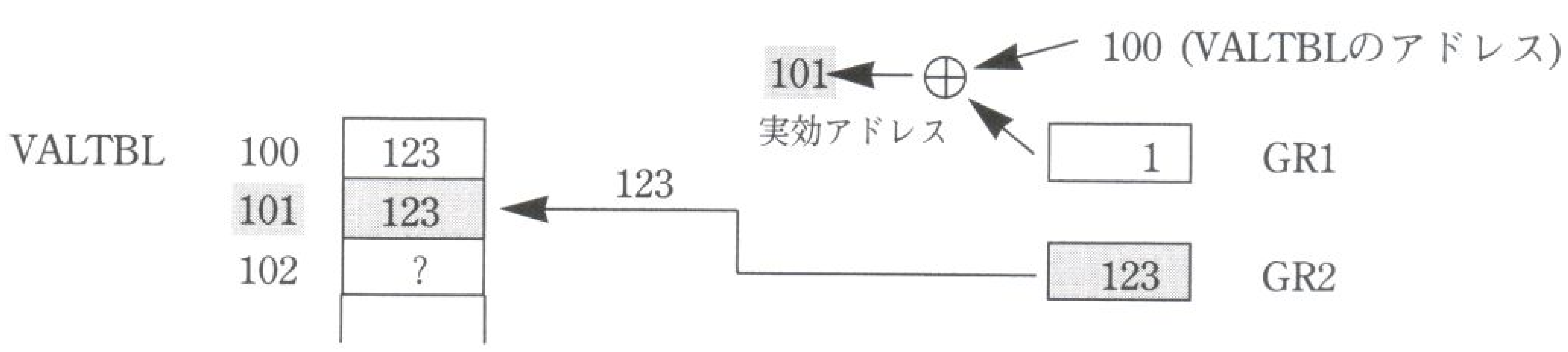

この時点で、「ST　GR2,VALTBL,GR1」を実行したとします。この命令は、GR1の値がゼロですからVALTBLの先頭の語にGR2の値を格納することを意味します。この動作を次に示します。




こののち、GR1の値を1つカウントアップします。




この状態で、再び「ST　GR2, VALTBL, GR1」を実行したとします。これは、GR1の値が1ですからVALTBLの先頭の次の語にGR2の値を格納することを意味します。この動作を次に示します。




このようにGR1の値を1つずつ19まで動かしながら、GR2の値をテーブルVALTBLに格納していきます。

## 演習問題11-2-1

プログラムVALSET1では、指標レジスタを0,1,2,...,19と動かしています。テーブルの個数は20個なのですから、指標レジスタは1,2,3,...,20とすべきではないでしょうか。

### 11.2.2 テーブルに値を下からセットする方法：カウントダウン

**例題 11-2**

例題11−1で作成したプログラム(VALSET1)は、カウンタとなるレジスタGR1の値を0,1,2,...,19と増やしていきました。これを、今度は、カウンタを19, 18, 17, ..., 2, 1, 0とカウントダウンしていく方法でプログラムを作成してください(VALSET2.casl)。

プログラムを示します。カウンタGR1の値には最初20をセットしておき、ループの先頭で一つカウントダウンします。この処理によって、GR1の値を1減らしていますので、最初にループの本体を処理するときには19となるのです。そして、そのカウンタGR1の値をループのたびごとに1つずつ減じていき、GR1の値が負になったらループを抜けるようにしています。

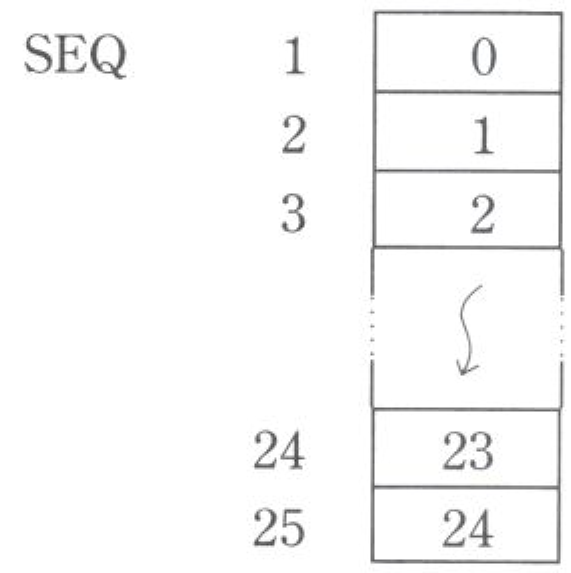
```
VALSET2   START
          LAD     GR1,20            GR1:カウンタ(19,18,17,...,2,1,0と動かす)
          LAD     GR2,123           GR2:データ123
          LAD     GR3,1             GR3(=1):定数1(カウントダウン用)
LOOP      SUBA    GR1,GR3           GR1をカウントダウン
          JMI     EXITLOOP          GR1が負ならばループを抜ける
          ST      GR2,VALTBL,GR1    GR2(=123)→VALTBLのGR1番目
          JUMP    LOOP              ループを続行
EXITLOOP  RET                        ;プログラムの終了
VALTBL    DS      20                20個分のデータを確保
          END
```

## 演習問題11-2-2

(1) データを順番に格納する

25語のテーブルSEQがあります。次のプログラムは、テーブルSEQに0,1,2,3,..24を格納するプログラム(SEQSET.casl)です。このプログラムの実行後、テーブルSEQは次のようになります。

```
SEQ   DS   25
```




空欄を適当な命令で埋めてください。

ヒント：このプログラムのポイントは、テーブルSEQのGR1番目に転送するデータを汎用レジスタGR1そのものとしている点です。

```
SEQSET      START
            LAD       GR1, [ 1 ]      GR1:カウンタ(24,23,22,...,2,1,0と動かす)
            LAD       GR3,1           GR3(=1):定数1(カウントダウン用)
LOOP        SUBA      GR1,GR3         GR1をカウントダウン
            [ 2 ]     EXITLOOP        GR1が負ならばループを抜ける
            ST        [ 3 ]           GR1→SEQのGR1番目
            JUMP      LOOP            ループを続行
EXITLOOP    RET                       ;プログラムの終了
SEQ         DS        25              25個分のデータを確保
            END
```

(2)

プログラムSEQSETでは、SEQに0,1,2,3...,24と格納していましたが、これを1,2,3,...,24,25と格納するプログラムがSEQSET1です。空欄に適当な字句を入れてください。

```
SEQSET1     START
            LAD      GR1,25           GR1:インデックス(24,23,...,2,1,0と動かす)
            LAD      GR2,26           GR2:この値が下から格納されていく。すぐに−1
                                      ;されるので初期値は26になっている
            LAD      GR3,1            GR3(=1):定数1(カウントダウン用)
LOOP        SUBA     GR1,GR3          GR1をカウントダウン
            JMI      EXITLOOP         GR1が負ならばループを抜ける
            [          1          ]   GR2をカウントダウン
            ST       [ 2 ],SEQ,GR1    GR2→SEQのGR1番目
            JUMP     LOOP             ループを続行
EXITLOOP    RET                       ;プログラムの終了
SEQ         DS       25               25個分のデータを確保
            END
```

## 11.3 テーブル集計

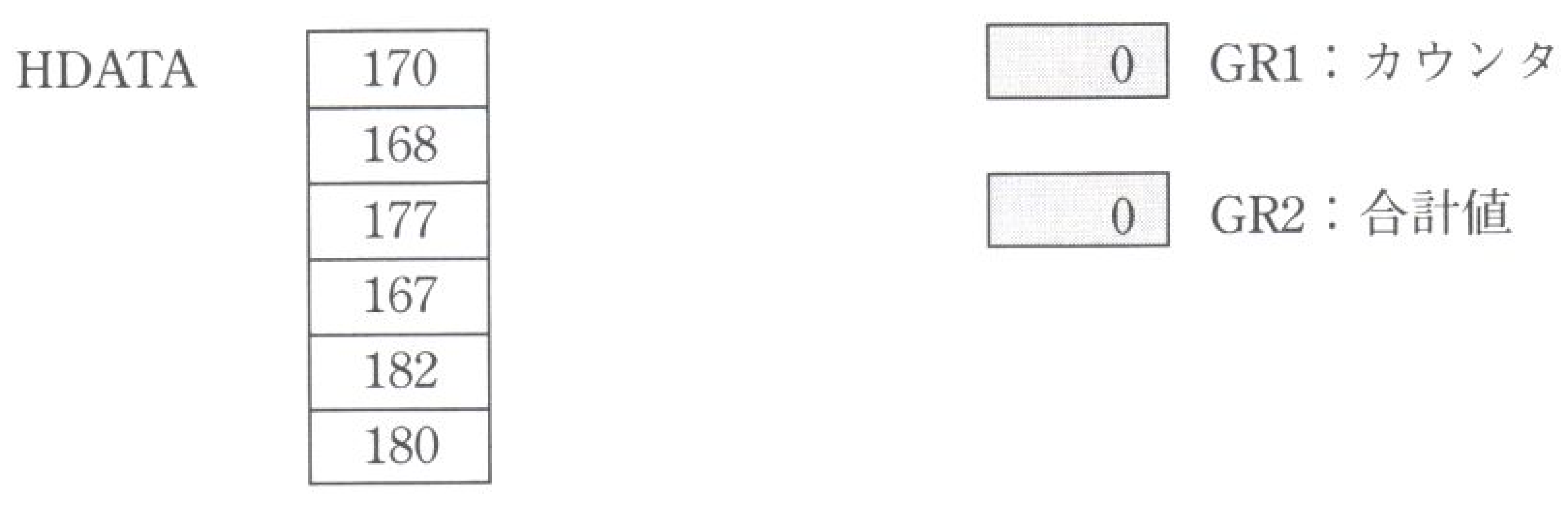
テーブルの中のデータを加えたり、何らかの加工をして加えたりするプログラムを**テーブル集計**といいます。

> **例題 11-3**
>
> テーブルHDATA(6件)中のデータ(身長データ)の合計を求め、それをSに格納するプログラム(TBLSUM.casl)を作成してください。

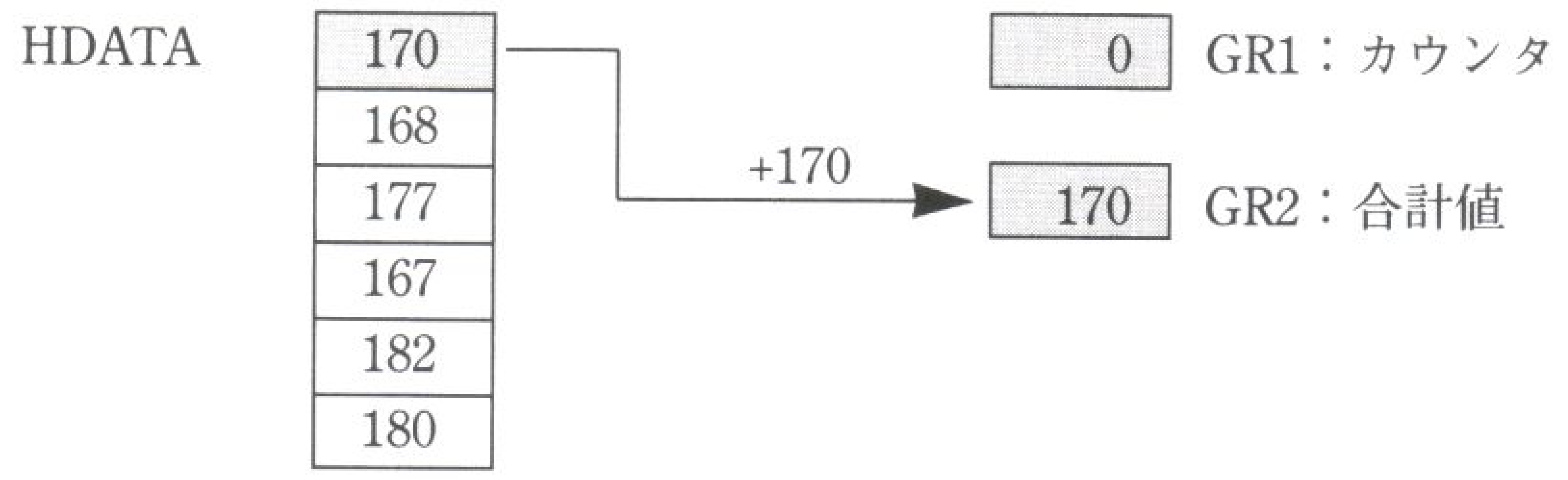
まず、最初にプログラムを示します。

```
TBLSUM   START
; 初期設定 ==========================================================
         XOR     GR1,GR1          GR1:カウンタ(0,1,2,..,5と動かす)
         XOR     GR2,GR2          GR2:合計値、ゼロクリア
         LAD     GR3,1            GR3(=1):定数1(カウントアップ用)
         LAD     GR4,6            GR4(=6):語数=カウンタ上限-1
; 合計値を求めるループ ================================================
LOOP     ADDA    GR2,HDATA,GR1    GR2(合計値)にVALTBLのGR1番目を加える
         ADDA    GR1,GR3          GR1をカウントアップ
         CPA     GR1,GR4          GR1と6を比較
         JMI     LOOP             GR1<6ならLOOPへ
; 合計値を保存 ==============================================
         ST      GR2,S            合計値→S
         RET                      ;プログラムの終了
HDATA    DC      170,168,177,167,182,180   身長データ
S        DS      1
         END
```

このプログラムについて説明しましょう。初期設定が終了した時点でプログラムは次のようになっています。



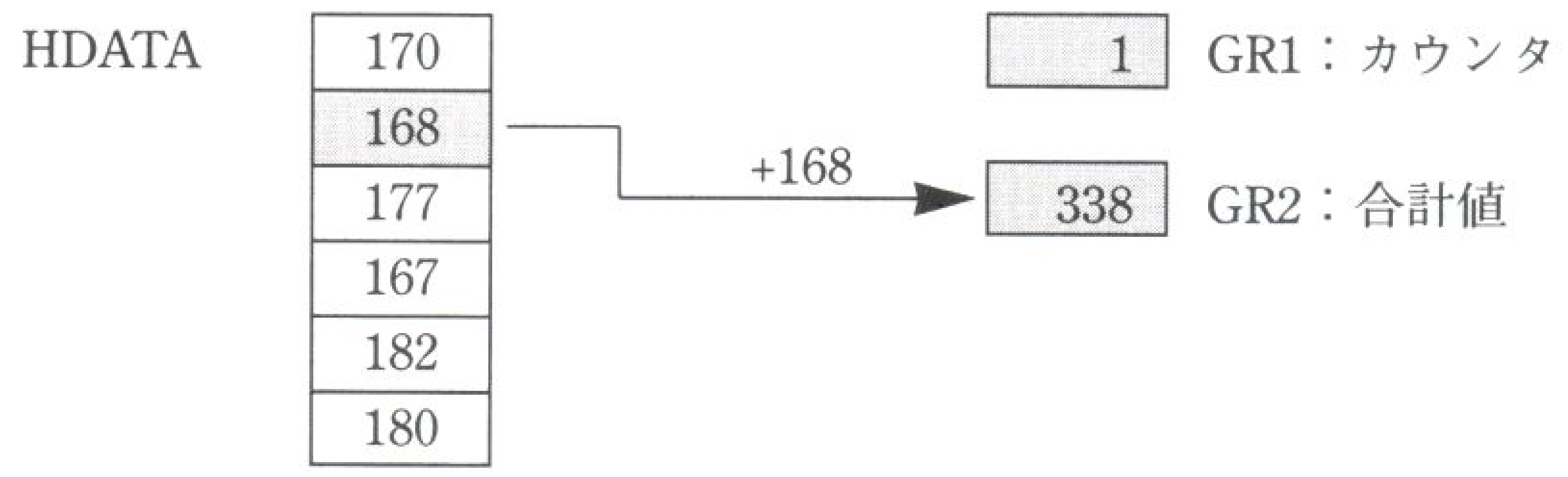

この状態で、「ADDA GR2,HDATA,GR1」を実行します。GR1の値がゼロですから、「HDATA,GR1」によって生成される実効アドレスは、最初のデータが格納された語のアドレスです。このADDA命令を実行すると、次のようになります。




そして、カウンタGR1の値を+1します。




この状態で、「ADDA GR2,HDATA,GR1」を実行します。GR1が1ですから、「HDATA,GR1」によって生成される実効アドレスは、2番目のデータが格納された語のアドレスになります。ADDA命令を実行すると、次のようになります。




さらに、カウンタGR1の値を1ずつ増やしていき、テーブル中の177, 167, 182, 180の値をGR2に加えて、値を集計します。なお、ここに示すプログラムの実行の様子は、本書に付属のCASLDVでグラフィカルに見ることができます。ぜひ、試してください。

## 演習問題11-3

次のプログラム(TBLSUM1.casl)は、TBLSUMを変更して、データの最後にあるゼロを判定し、ゼロが出てくるまで集計し、人数をNに、合計をSに格納するプログラムです。空欄を適当な命令で埋めなさい。

```
TBLSUM1   START
          XOR      GR1,GR1     GR1:カウンタ(0,1,2,..と動かす)
          XOR      GR2,GR2     GR2:合計値、ゼロクリア
          LAD      GR3,1       GR3(=1):定数1(カウントアップ用)
          [  1  ]  GR4,GR4     GR4:人数、最初は0人
LOOP      [       2       ]    GR5←HDATAのGR1番目
          [  3  ]  FINISH      もし、それがゼロならばFINISHへ
                   ADDA        GR2,  [  4  ] GR2(合計値)にGR5の値を加える
          ADDA     GR1,GR3     GR1をカウントアップ
          ADDA     GR4,[  5  ] GR4をカウントアップ
          JUMP     LOOP        LOOPへ無条件ジャンプ
FINISH    ST       GR2,S       合計値→S
          ST       GR4,N       人数→N
          RET                   ;プログラムの終了
HDATA     DC       170,168,177,167,182,180,0   身長データ
S         DS       1                        ↑
N         DS       1                        |
          END                          データの最後の印
```

## 11.4 テーブルサーチ(表検索)

テーブルの中から特定の数値(文字の場合もあります)を見つけたり、特定の条件に合うデータの個数を求める処理のアルゴリズムの総称を**テーブルサーチ**(あるいはテーブル検索、表検索)といいます。

たとえば、次のテーブルがあるとします。

```
MTABLE     DC    458,-256,45,-58,727,333
```

この中から、たとえば、−58を見つけ出すという処理がテーブルサーチです。この場合、テーブルMTABLE中にある−58の位置を求めます。先頭のデータを0番目と数えるので先頭の位置で見つかった場合には0、2番目の位置で見つかった場合には1が求める位置となります。上記の例では、求める位置は3となります。もし、見つからなかった場合には、位置情報としては意味のない−1を「位置」とします。したがって、−1という位置は対象データがテーブル中にないことを意味します。

直感的に考えれば、この処理は、テーブルの先頭からデータを順次取り出し、−58と比較していけばよいわけです。あるいは、テーブルの最後から前に向かってデータを取り出していっても同じです。前から検索しても、後ろから検索しても、テーブル中にデータがn個あるとすると、検索するデータがテーブル中にある場合には平均してn/2回の検索回数が必要になります。もし、検索するデータがテーブル中に無い場合には、全部検索しなければならないので、n回の検索が必要になります。この検索方法を**線形探索**と呼びます。

これに対して、データが次のように小さい順に並んでいることが分かっている場合には、どうしたらよいでしょうか。

```
MTABLE     DC    -501,-256,-58,45,333,458,727
```

この場合、データ数が7個ですから、まず真中のデータを調べます。テーブルを扱う場合、0番目から勘定した方が分かりやすいので、最後の6番目(727)の番号6を2で割り(右に1ビットシフト)、3を得ます。3番目のデータは45です。これがちょうと真中のデータということになります。

```
                    0    1    2  3   4    5    6  ←何番目か
MTABLE     DC    -501,-256,-58, 45 ,333,458,727
                 |←─────────→|  ↑
                 -58がある範囲    |
                                まん中
```

さて、−58は45よりも小さいですから、−58はこのデータ45の左側にあることが分かります。そこで、今度は、0番目から2番目のまん中を求めます。2を1ビット右にシフトすると、1になります。1番目のデータは−256です。このことから、−58は1番目よりも左で3番目よりも右にあることがわかります。それは2番目しかありません。2番目のデータは−58なので検索しているデータが見つかりました。この場合には、わずか3回の検索回数で見つかりました。

このように次々に半分にして検索する方法を**二分探索**と呼びます。ただし、この方法は、検索効率が良いのですが、元のデータがソート(*1)されていることが必要になります。

以上の事柄を表11-3にまとめます。

(*1) ソートとはデータを大きい順、あるいは、小さい順に並び替える処理のことを言います。

表11-3 テーブルサーチの効率(＊1)

| | 平均探索回数 | 検索失敗時の検索回数 | データの前処理 |
|---|---|---|---|
| 線形探索(上から下へ) | n / 2 | n | 不要 |
| 線形探索(下から上へ) | n / 2 | n | 不要 |
| 二分探索 | $\lfloor \log_2 n \rfloor + \frac{1}{2}$ | $\lfloor \mathrm{lon}_2 n \rfloor + 1$ | ソートが必要 |

### 11.4.1 テーブル検索：線形探索

**例題 11-4**

テーブルMTABLEの中から値－58を検索するプログラム(FOUNDT.casl)を作成してください。ただし、テーブル中のデータはランダムに並んでいるものとします。

データがランダムに配置されているので線形探索になります。カウンタGR1の値を0から5まで次々に動かしながら、「CPA　GR2,MTABLE,GR1」でGR2の値(見つける値)とテーブル中の値を比較し、一致すればループを終了しています。しかし、このプログラムFOUNDTは演習問題11-4-1にも示していますが、ループの中でジャンプが複雑にからみあっていてあまり良いプログラムとはいえません。

```
FOUNDT    START
          XOR    GR1,GR1              GR1:カウンタ(0,1,2,..,5と動かす)
          LD     GR2,FVALUE           GR2:見つける値
          LAD    GR3,1                GR3(=1):定数1(カウントアップ用)
          LAD    GR4,6                GR4(=6):語数=カウンタ上限-1
          LAD    GR5,-1               GR5:見つかった位置(-1:見つからなかった)
LOOP      CPA    GR2,MTABLE,GR1       検索値とMTABLEのGR1番目を比較
          JNZ    SKIP                 検索値(GR2)と等しくなければ何もしない
          LD     GR5,GR1              カウンタの値を「GR5:見つかった位置」へ
          JUMP   SETV                 ループを抜ける
SKIP      ADDA   GR1,GR3              GR1をカウントアップ
          CPA    GR1,GR4              GR1と6を比較
          JMI    LOOP                 GR1<6ならLOOPへ
SETV      ST     GR5,FPOS             位置をFPOSへ
          RET                         ;プログラムの終了
MTABLE    DC     458,-256,45,-58,727,333  6個分のデータを確保
FVALUE    DC     -58
FPOS      DS     1
          END
```

しかし、アセンブラは分岐命令とラベルでさまざまな処理が構築できてしまうので、このような複雑なロジックのプログラムであっても理解できなければいけません。また、このプログラムと演習問題11-4-1のプログラムとの違いも十分に理解しておく必要があります。

(＊1) 二分探索の中で⌊x⌋(xのフロアー)は、xより小さいか等しい整数のうち最大のものです。この式の導き方は、たとえば、Sara Basse著、岩野和生、加藤直樹、永持仁訳、アルゴリズム入門ー設計と解析ー、アジソン・ウェスレイ・パブリッシャーズ・ジャパンなどを参考にしてください。

## 演習問題11-4-1

プログラムFOUNDTは、ループの構造が次のようになっています。

```
→ LOOP    CPA     GR2,MTABLE,GR1
          JNZ     SKIP
          LD      GR5,GR1
          JUMP    SETV
→ SKIP    ADDA    GR1,GR3
          CPA     GR1,GR4
          JMI     LOOP
           ↓ループ終了
  SETV    ST      GR5,FPOS ←
```

つまり、ループの範囲の中にジャンプ命令(JNZ SKIP)が一つあります。そこで、FOUNDTのこの部分を、ループ内の処理は単なる直線的な流れにして、ループ内のジャンプ命令を1つだけにしたプログラムを作りました(FOUNDT1.casl)。変更した部分だけを以下に示します。空欄に適当な命令を入れてください。

```
FOUNDT1 START
          ∫
LOOP    CPA     GR2,MTABLE,GR1     検索値とMTABLEのGR1番目を比較
        [          1          ]     検索値(GR2)と等しければループを脱出
        ADDA    GR1,GR3            GR1をカウントアップ
        CPA     GR1,GR4            GR1と6を比較
        JMI     LOOP               GR1<6ならLOOPへ
SETV    ST      GR5,FPOS           位置をFPOSへ
        RET                         ;プログラムの終了
FOUND   [          2          ]     カウンタの値を「GR5:見つかった位置」へ
        JUMP    [      3      ]
          ∫
        END
```

プログラムFOUNDTよりもFOUNDT1の方が流れがすっきりしていて理解しやすいと思います。

### 11.4.2 特定の条件に合うデータの個数のカウント

**例題 11-5**

テーブルの中から偶数のデータを検索し、その個数を求めるプログラム(COUNTT.casl)を作成してください。

データが偶数か奇数か判断するには、1と論理積をとれば良いのでした(復習)。これは、データが負の場合でも成立します。

```
COUNTT  START
        XOR    GR1,GR1          GR1:カウンタ(0,1,2,..,5と動かす)
        XOR    GR2,GR2          GR2:偶数の個数、最初はゼロ
        LAD    GR3,1            GR3(=1):定数1(ビットマスクとカウントアップ両用)
        LAD    GR4,6            GR4(=6):語数=カウンタ上限-1
LOOP    LD     GR5,MTABLE,GR1   GR5←MTABLEのGR1番目
        AND    GR5,GR3          GR5(MTABLEのGR1番目)の最下位ビットを抽出
        JNZ    SKIP             ゼロでない==>奇数なのでSKIPする
        ADDA   GR2,GR3          GR2:偶数の個数をカウントアップ
SKIP    ADDA   GR1,GR3          GR1をカウントアップ
        CPA    GR1,GR4          GR1と6を比較
        JMI    LOOP             GR1<6ならLOOPへ
SETV    ST     GR2,N            個数をNへ
        RET                     ;プログラムの終了
MTABLE  DC     458,-256,45,-58,727,333  6個分のデータを確保
N       DS     1
        END
```

## 演習問題11-4-2

プログラムCOUNTTで、LOOP～SKIPの部分の効率を良くしようとして、次のようにプログラムを変更しました。正しく動作するでしょうか。

```
LOOP    AND    GR3,MTABLE,GR1   MTABLEのGR1番目の最下位ビットを抽出
        JNZ    SKIP             ゼロでない==>奇数なのでSKIPする
        ADDA   GR2,GR3          GR2:偶数の個数をカウントアップ
SKIP    ADDA   GR1,GR3          GR1をカウントアップ
```

### 11.4.3 最大値・最小値の発見

**例題 11-6**

テーブルの中から最大値を検索するプログラム(MAXV.casl)を作成してください。

プログラムの概略は次のようになります。テーブルの先頭の語の値(以下の例では、458)を、とりあえず最大の値と考え、レジスタGR2にセットします。そして、テーブルの残りの語の値とGR2の値を次々に比較していきます。もし、テーブルの語の値の方が、GR2の値よりも大きければ、テーブルの語の値をGR2にロードしGR2の値を更新します。つまり、GR2にはその時点での最大値が常に格納されているようにするのです。テーブルをすべて検索し終えたときには、GR2には、テーブルの最大値が格納されています。

なお、テーブルの最初の語を最大値として、GR2に最初に格納しているため、テーブルの検索は、テーブルの2番目からでよいことに注意してください。そのため、指標レジスタGR1の値は0ではなく、1から始まっています。

```
MAXV    START
        LAD    GR1,1              GR1:カウンタ(1,2,..,5と動かす)
        LD     GR2,MTABLE         GR2:最大値、初期値はMTABLEの先頭要素の値(この例では458)
        LAD    GR3,1              GR3(=1):定数1(カウントアップ用)
        LAD    GR4,6              GR4(=6):語数=カウンタ上限-1
LOOP    LD     GR5,MTABLE,GR1     GR5←MTABLEのGR1番目
        CPA    GR2,GR5            それまでの最大値と比較
        JPL    SKIP               それまでの最大値(GR2)が大きければ何もしない
        LD     GR2,GR5            新しい要素を最大値とする
SKIP    ADDA   GR1,GR3            GR1をカウントアップ
        CPA    GR1,GR4            GR1と6を比較
        JMI    LOOP               GR1<6ならLOOPへ
        ST     GR2,MAX            最大値をMAXへ
        RET                       ;プログラムの終了
MTABLE  DC     458,-256,45,-58,727,333  6個分のデータを確保
MAX     DS     1
        END
```

ただし、上記のプログラム(MAXV)は、GR2>GR5のときにラベルSKIPへジャンプしていますので、GR2≦GR5のときに最大値の更新を行っています。この処理の欠点は、GR2=GR5のときも最大値の更新を行ってしまうことです。そこで次のような改良を行うことができます。

```
        CPA    GR2,GR5
        JPL    SKIP
        LD     GR2,GR5
SKIP    ADDA   GR1,GR3
```

もとのプログラム

改良(1)

```
        CPA    GR2,GR5
        JPL    SKIP
        JZE    SKIP
        LD     GR2,GR5
SKIP    ADDA   GR1,GR3
```

GR2=GR5のときにもSKIPへ

改良(2)

```
        CPA    GR5,GR2
        JPL    UPDATE
NEXT    ADDA   GR1,GR3
          ⋮
        RET
UPDATE  LD     GR2,GR5
        JUMP   NEXT
```

GR5>GR2のときにUPDATEにジャンプしてGR2を更新する更新

改良(1)では、GR2=GR5の条件のときもSKIPへジャンプするようにしています。改良(2)はロジックを反転させて更新時(GR5>GR2)にUPDATEラベルにジャンプしています。ただし、この場合、ジャンプ先(UPDATE)から「JUMP NEXT」でNEXTラベルへ戻る必要があります。

## 演習問題11-4-3

(1) なぜ、「LD GR2,MTABLE」でテーブルMTABLEの先頭要素の値がGR2に格納されるのですか。

(2) MAXV(改良(1))をベースにして、最小値を求めるプログラム(MINV.casl)を作りました。レジスタGR1〜GR4にセットされている値、およびMTABLEの内容は、MAXVと全く同じです。最小値を格納する領域がMINとなっています。LOOP以下を示します。空欄に適当な命令を入れてください。

```
LOOP    LD      GR5,MTABLE,GR1   GR5←MTABLEのGR1番目
        CPA     GR2,GR5          それまでの最小値と比較
        [ 1 ]   SKIP             それまでの最小値(GR2)が小さければ何もしない
        [ 2 ]   SKIP             等しいときも更新しない
        LD      GR2,GR5          新しい要素を最小値とする
SKIP    ADDA    GR1,GR3          GR1をカウントアップ
        CPA     GR1,GR4          GR1と6を比較
        JMI     LOOP             GR1<6ならLOOPへ
        ST      GR2,MIN          最小値をMAXへ
        RET
```

(3) プログラムMAXVで、テーブルの検索を2番目からしていますが、先頭の語から検索したらどうなりますか。

### 11.4.4 データの多重サーチ

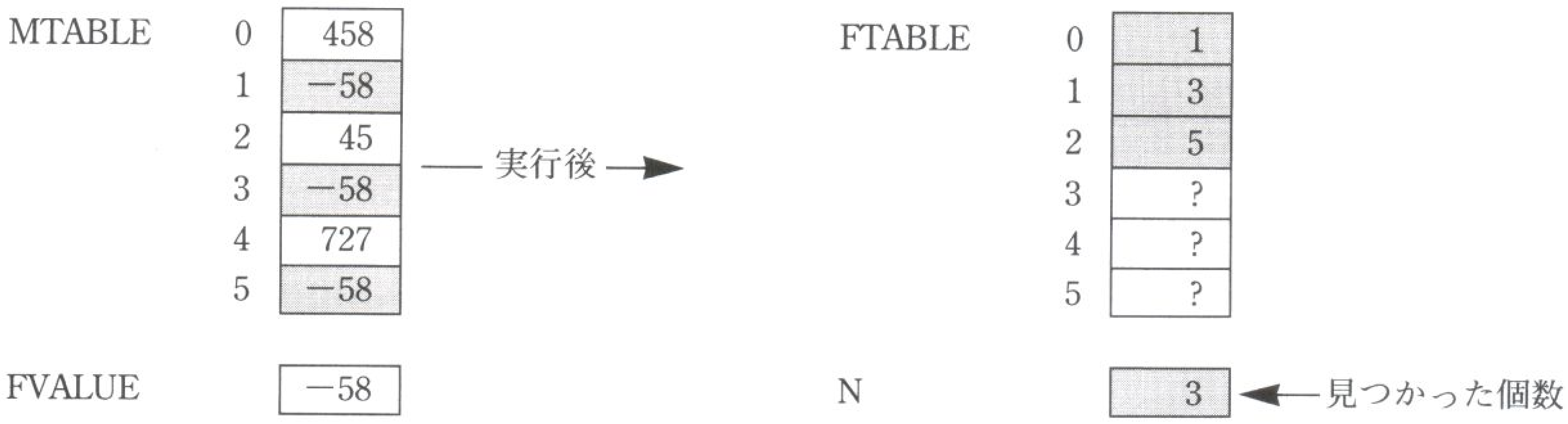
**例題 11-7**

テーブル中にある値が複数個ある場合を考えます。たとえば、次のテーブルMTABLEには、−58という数値が3つあります。

```
MTABLE  DC    458,-58,45,-58,727,-58   6個分のデータを確保
```

この−58という数値の位置(1,3,5)(先頭を0と数える)を，テーブルFTABLEに格納し、格納された個数をNにセットするプログラム(FOUNDW.casl)を作成してください。

検索する値はFVALUEに格納されています。プログラム実行後、テーブルFTABLEの内容は次のようになります。テーブルMTABLEとFVALUEの値は変わりません。



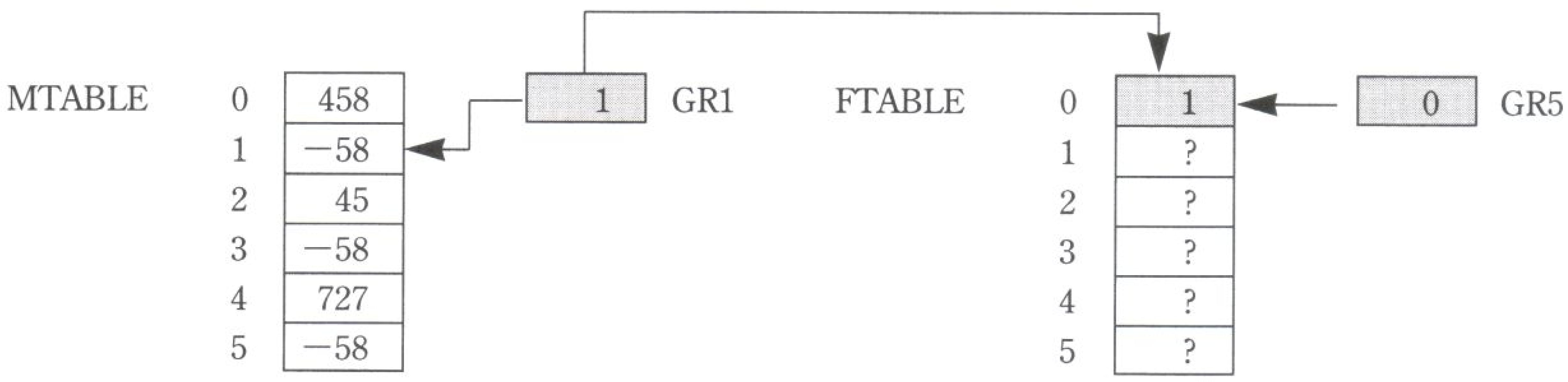

これを実現するには、両方のテーブルに対し、それぞれ、指標レジスタを1つずつ用意します。テーブルMTABLEに対してはGR1、テーブルFTABLEに対してはGR5を用いることにします。そして、両方(GR1とGR5)とも初期値を0にしておきます。これで、レジスタGR1はテーブルMTABLEの先頭の語、レジスタGR5はテーブルFTABLEの先頭の語を指すことになります。




レジスタGR1の値を動かしながら、テーブルMTABLEの中を検索し、−58を見つけます。この場合、GR1の値が1のときにテーブルMTABLEの値が−58になります。




ここで、レジスタGR1の値を、テーブルFTABLEのGR5番目(=0)に格納します。この処理は、「ST GR1, FTABLE, GR5」という1つの機械語命令で実現できます。




次に、レジスタGR5を1つ増やします。



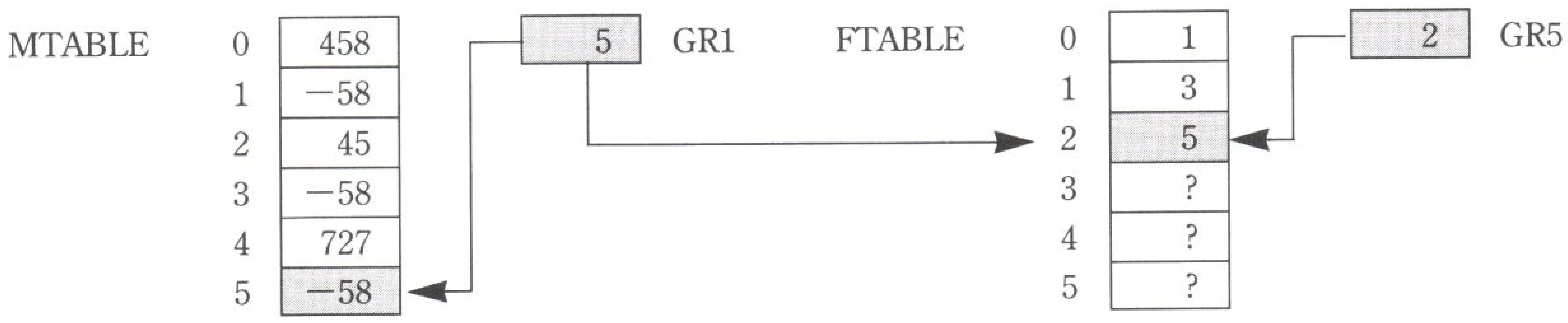
この状態でさらにMTABLEを探すと、3番目に−58が見つかります。そこで、GR1の値をFTABLEにストアします。



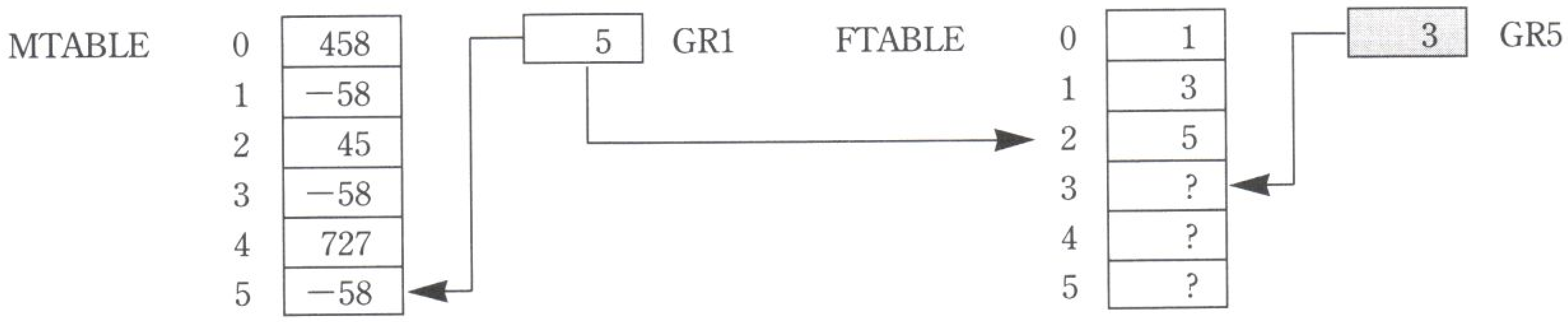

次に、GR5の値を1つ増やします。




さらに、MTABLEを探すと5番目にも−58が見つかりました。そこで、GR1の値をFTABLEにストアします。




そして、GR5の値を1つ増やします。




ここで、MTABLEの処理が終わります。さて、このとき、レジスタGR5は、FTABLEに格納された最後のデータの次を指していますが、この値は、FTABLEに格納されたデータの個数と考えることができます。このGR5の値をNに格納します。

以上の流れをプログラム化したものが次のプログラムです。

```
FOUNDW   START
         XOR  GR1,GR1               GR1:カウンタ(0,1,2,..,5と動かす)
         LD   GR2,FVALUE            GR2:見つける値
         LAD  GR3,1                 GR3(=1):定数1(カウントアップ用)
         LAD  GR4,6                 GR4(=6):語数=カウンタ上限-1
         XOR  GR5,GR5               GR5:「見つかった個数」(0:見つからなかった)
LOOP     CPA  GR2,MTABLE,GR1        検索値とMTABLEのGR1番目を比較
         JZE  FOUND                 検索値(GR2)と等しければループを脱出
ENTERLOP ADDA GR1,GR3               GR1をカウントアップ
         CPA  GR1,GR4               GR1と6を比較
         JMI  LOOP                  GR1<6ならLOOPへ
         ST   GR5,N                 見つかった個数をFPOSへ
         RET                        ; プログラムの終了
FOUND    ST   GR1,FTABLE,GR5        カウンタの値をFOUNDのGR5番目へ
         ADDA GR5,GR3               GR5:「見つかった個数」をカウントアップ
         JUMP ENTERLOP              ループに戻る
MTABLE   DC   458,-58,45,-58,727,-58          6個分のデータを確保
FTABLE   DS   6
FVALUE   DC   -58
N        DS   1
         END
```

## 11.5 編集処理

**編集処理**とは、計算結果などをディスプレイ画面やプリンタに表示する際に、見やすい形式に整えることを言います。今、12915というデータをディスプレイに表示することを考えましょう。この数値は、2進数で表現すれば、$0011001001110011_{(2)}$となります。

この数値をそのままディスプレイに出力すると不思議なことにｓと表示されてしまいます。次のプログラムは、このことを証明するものです(BINOUT.casl)。

```
BINOUT  START
        LD      GR1,=12915
        ST      GR1,OUTDAT
        OUT     OUTDAT,OUTLEN
        RET
OUTLEN  DC      1
OUTDAT  DS      256
        END
```

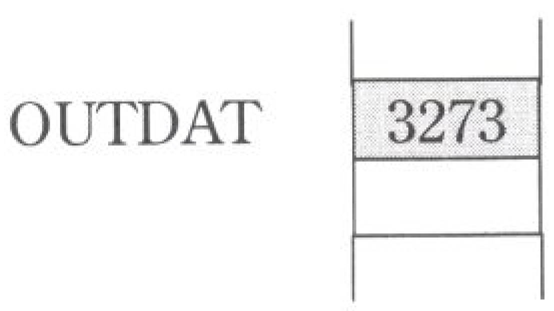
```
BINOUT  START
        OUT  =12915,=1
        RET
        END
```

(リテラルを用いた例)

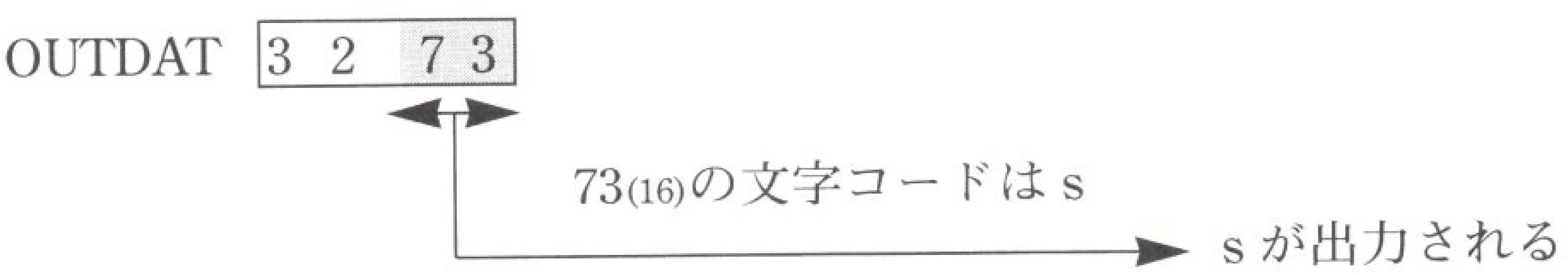
これを実際に実行すると、次の結果を得ることができます。

```
s
```

これは、なぜでしょう。その理由は文字コードにあります。まず、このプログラムを詳細にトレースしてみましょう。まず、レジスタGR1に12915が格納されます。これは16進数では、$3273_{(16)}$です。そして、それが、OUTDATの先頭に配置されます。




さて、OUT命令は、テーブルの内容を、指定された文字数だけ出力する命令でした。ここでは、出力文字数(OUTLEN)には1が格納されているので、1語分しか出力されません。1語分といっても、その1語の下位1バイトだけが文字コードとして出力されるのです。上位1バイトは無視されます。下位1バイトである$73_{(16)}$に相当する文字コードはｓですから、それが出力される訳です。




それでは、12915と表示するには、どのようにしたら良いのでしょうか。結論から言うと、この数値($3273_{(16)}$)を次のように変換します。

```
OUTDAT    0031  '1'の文字コード
          0032  '2'の文字コード
          0039  '9'の文字コード
          0031  '1'の文字コード
          0035  '5'の文字コード
```

これを確認するプログラムが次のものです(DATOUT.casl)。

```
DATOUT    START
          OUT     OUTDAT,OUTLEN
          RET
OUTLEN    DC      5
OUTDAT    DC      #0031
          DC      #0032
          DC      #0039
          DC      #0031
          DC      #0035
          END
```

実際に実行すると次のようになります。

```
12915
```

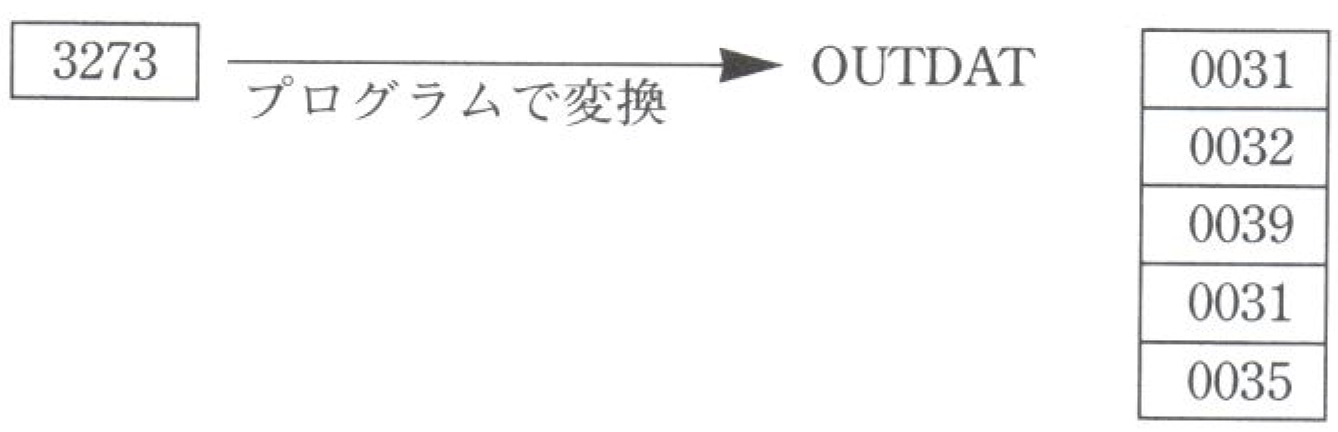
なお、プログラム(DATOUT)のOUTDATの定義の部分は、次のようにすることもできます。

```
OUTDAT    DC      '1'
          DC      '2'
          DC      '9'
          DC      '1'
          DC      '5'
```

あるいは、次のように1行にしてもOKです。

```
OUTDAT    DC      '12915'
```

要するに、数値データの12915は文字データの'12915'に変換しなければ人間が見てわかる形式では出力できないということです。




なお、リテラルを使うとOUT命令は次のように書くこともできます(最初に示したプログラム参照)。

```
OUT  ='12915',=5
```

### 11.5.1 内部データ(2進数) → 表示データ(10進数文字データ)

> **例題 11-8**
>
> 領域Aに格納されたデータを、ディスプレイに出力して表示するプログラム(BINDEC.casl)を作成してください。負のデータも扱えるようにしてください。

領域Aのデータをレジスタ GR0にロードして、GR0の内容を編集して出力するようにしましょう。今、仮にGR0には、−13547が格納されているとします。この値は16進数ではCB15(16)になります。

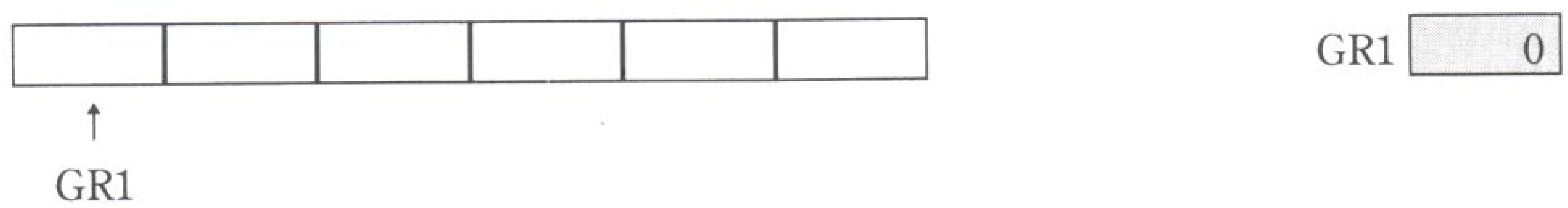
| | 16進数 | | 10進数 |
|---|---|---|---|
| GR0 | CB15 | GR0 | −13547 |

この値を対応する次の文字コードに変換するわけです。今までテーブルは縦に記述していましたが、紙面を使うので、この例では横で表現します。右側方向にアドレスが増加するように配置しますので注意してください。

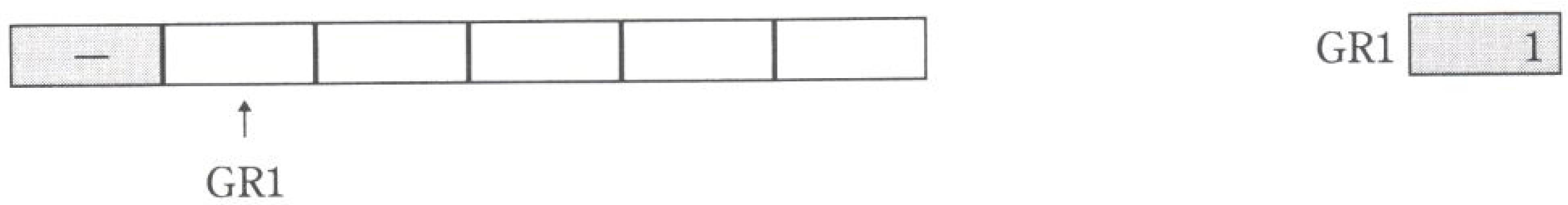
→アドレスの増加方向

| 文字コード表現 | − | 1 | 3 | 5 | 4 | 7 |
|---|---|---|---|---|---|---|
| 16進表現 | 002D | 0031 | 0033 | 0035 | 0034 | 0037 |

この手順は、思ったよりも単純です。まず、レジスタGR1がテーブルの先頭語を指すようにします(つまり、GR1にテーブルの先頭語からの変位0を格納します)。




もし、データが負の数ならば、GR1の位置に'−'(負符号)をセットし、GR1を+1します。



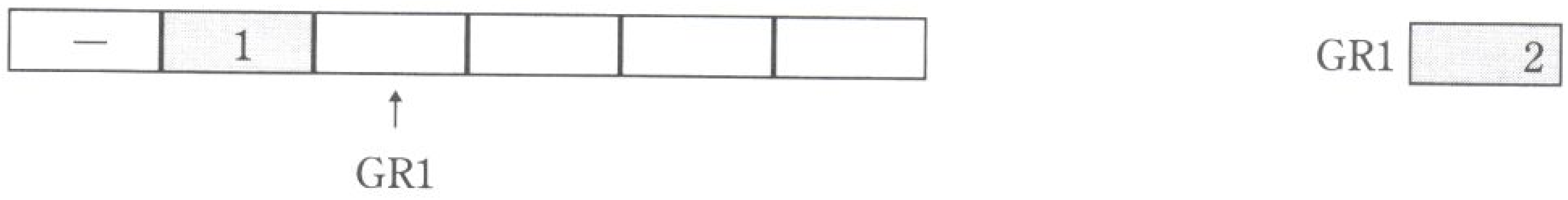

そして、データが負の場合には、GR0の補数を取って、正の数にします。データが正の場合には何もしません。この場合、GR0の値は負ですから、補数を取る必要があります。

| | 16進数 | | 10進数 | | 2進数 | |
|---|---|---|---|---|---|---|
| GR0 | CB15 | GR0 | −13547 | GR0 | 1100101100010101 | もとの数 |
| | ↓ | | ↓ | | ↓ | |
| GR0 | 34EA | GR0 | 13546 | GR0 | 0011010011101010 | ビット反転 |
| | ↓ | | ↓ | | ↓ | |
| GR0 | 34EB | GR0 | 13547 | GR0 | 0011010011101011 | +1して2の補数表現 |

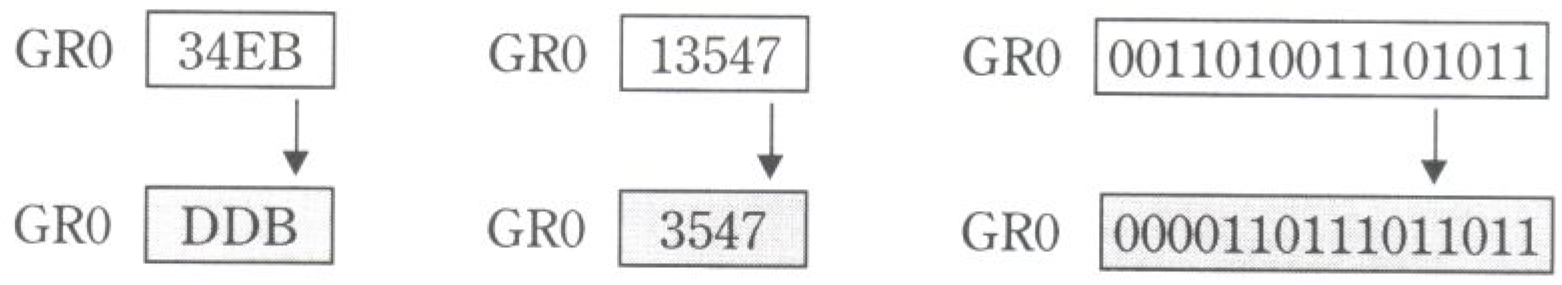
さて、いよいよ、GR0のデータを文字列へ変換していきます。まず、GR0のデータから10000を何

回引けるか調べます。つまり、10000で割って答えを求めます。この場合、その答えは1なのでGR1の位置に'1'をセットし、GR1を+1して、テーブルの次の語を指すようにします。

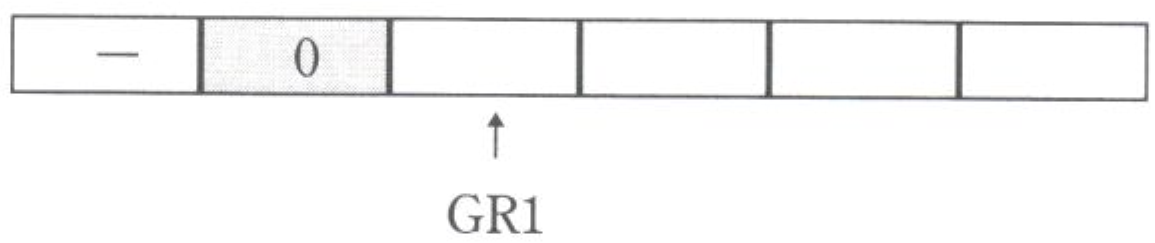
数値のコード1を文字の'1'に変換するには、1に0の文字コード'0'(0030(16))を加えます。1＋0030(16)=0031(16)となり、この値が'1'の文字コードになります。




GR0には、その余り(この場合、13547を10000で割った余りである3547)を残します。

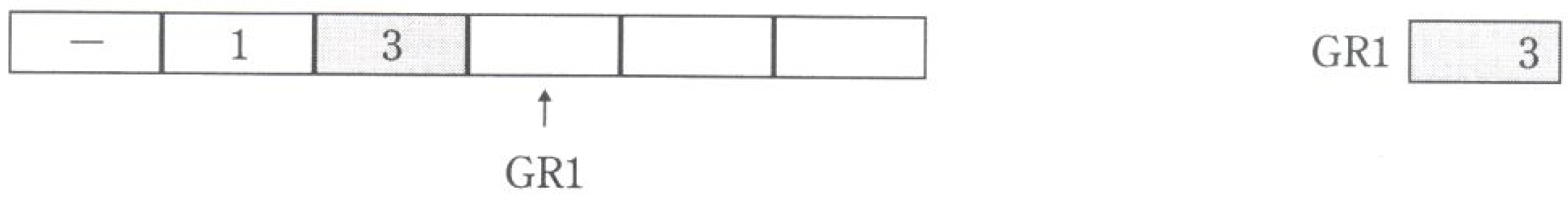



仮に、最初のGR0の値を10000で割った時の答えがゼロなら次のようになります。

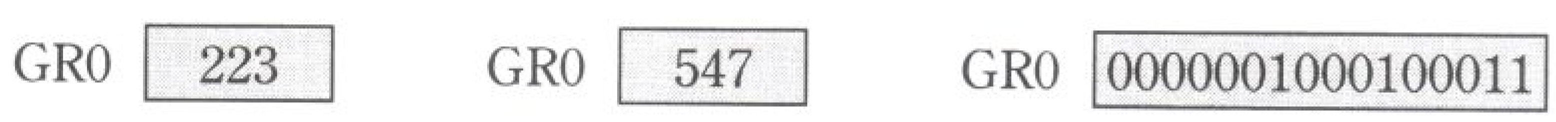



－0という表示は、表示としては見にくくなり、避けたいものです。これを避ける方法は、ゼロ抑制といい、総合問題11の演習11-4を参照してください。。

次に、残ったデータGR0から1000を何回引けるか調べます。つまり、1000で割って答えを求めます。この場合、その答えが3なのでGR1の位置に'3'をセットし、GR1を+1します。GR0には、その余りを入れます。



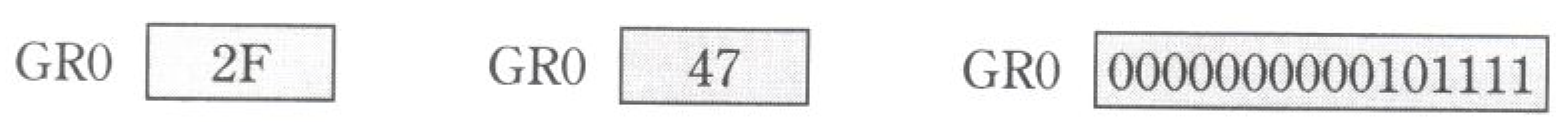

GR0の値を残りの547にします。




次に、残ったデータGR0から100を何回引けるか調べます。つまり、100で割って答えを求めます。この場合、その答えは5なのでGR1の位置に'5'をセットし、GR1を+1します。




GR0には、その余り47を入れます。



次に、残ったデータGR0から10を何回引けるか調べます。つまり、10で割って答えを求めます。この場合、その答えは4なのでGR1の位置に'4'をセットし、GR1を+1します。

| − | 1 | 3 | 5 | 4 | |
|---|---|---|---|---|---|
| | | | | | ↑ GR1 |

GR1 5

GR0には、その余り7を入れます。

GR0 7　GR0 7　GR0 0000000000000111

最後に残ったデータをGR1の位置にセットします。たとえば、この場合7なので'7'をセットします。

| − | 1 | 3 | 5 | 4 | 7 |
|---|---|---|---|---|---|
| | | | | | ↑ GR1 |

GR1 5

さて、ここで「10000で割る」、「1000で割る」、「100で割る」、「10で割る」という処理がありました。これは、10進数の各桁の重みを求めているわけです。この処理をそのまま、工夫せずに直線型のプログラムで書き下ろしてしまうとプログラムが長くなり、大変面倒です。そこで、ループを使ってこの処理を単純化しましょう。

この重み(10000,1000,100,10)をテーブルWTABLEに格納し、指標レジスタGR3を動かしながらチェックすることにしましょう。なお、次の例は、指標レジスタGR3をカウントダウンしているので重みが逆に配置されていることに注意してください。

```
                 GR3←4
→ CNVLOOP        GR3のカウントダウン
                 GR3が負になったらEXITLOOPへ    ループ脱出
                 GR4←WTABLEのGR3番目
  ループ         処理(GR0をGR4で割る)
                 JUMP  CNVLOOP

  WTABLE         DC    10,100,1000,10000
```

それでは、このプログラムを示します。

```
BINDEC   START
; 初期設定 ==================================================
         XOR    GR1,GR1        GR1:編集格納領域のインデックス
         LAD    GR2,1          GR2:定数1(計算用)
         LAD    GR3,4          GR3:重みテーブル(WTABLE)のインデックス
         LD     GR0,A          GR0:変換するデータ
; 負符号の処理 ==================================================
         JMI    MINUS          もし、変換するデータが負ならば'−'を付ける
; 数値編集のメインループ ==========================================
; 重み 10000, 1000, 100, 10で割った数値を文字に変えてDIGITにセット
CNVLOOP  SUBA   GR3,GR2        重みテーブルのインデックスをカウントダウン
```

```
         JMI     EXITLOOP         もし、すべての重みを計算したら終了し、1桁目を処理
         LD      GR4,WTABLE,GR3  GR4←重み(GR3番目)
         ; -----------------------------------------------------
         ; GR0をGR4で割った値(GR5)と余り(GR0)を求める
         XOR     GR5,GR5          除算の商をゼロにしておく
DIVLOOP  CPA     GR0,GR4          GR0とGR4(重み)を比較
         JMI     DIVFIN           GR0からGR4(重み)が引けなくなったら除算処理終了
         SUBA    GR0,GR4          GR0からGR4(重み)を引く
         ADDA    GR5,GR2          GR5(商)を1つカウントアップ
         JUMP    DIVLOOP          さらに除算処理を続ける
         ; -----------------------------------------------------
         ; 求めた桁(GR5)を文字コードに変換して、編集領域にセット
DIVFIN   ADDA    GR5,='0'         文字コード'0'を加えて、GR5を文字コード化する
         ST      GR5,DIGIT,GR1    文字に変換した桁を編集領域にセット
         ADDA    GR1,GR2          GR1(編集領域インデックス)をカウントアップ
         JUMP    CNVLOOP          次の桁の処理に移る
; 負符号の実際の処理 ===================================================
MINUS    LD      GR7,='-'         GR7←'-'
         ST      GR7,DIGIT,GR1    '-'を編集領域にセット
         ADDA    GR1,GR2          GR1(編集領域インデックス)をカウントアップ
         ; GR0の2の補数を求めて、符号を反転する
         XOR     GR0,=#FFFF       ビット反転
         ADDA    GR0,GR2          +1する
         JUMP    CNVLOOP          メインの処理に復帰する
; 最後の桁の処理 =======================================================
EXITLOOP ADDA    GR0,='0'         '0'を加えて最後の桁を文字の数値にする
         ST      GR0,DIGIT,GR1    それを編集領域にセット
         ADDA    GR1,GR2          GR1(編集領域インデックス)をカウントアップ(これで
                                  ;編集領域の文字数になる)
         ST      GR1,OUTLEN       GR1を文字数として、OUTLENにストア
         OUT     DIGIT,OUTLEN     編集領域を出力
         RET                      ; プログラムの終了
A        DC      -32145           変換する数値
WTABLE   DC      10,100,1000,10000  10進数の重み
DIGIT    DS      6                編集領域
OUTLEN   DS      1                編集領域の長さ
         END
```

## ポイント

数値を表示するには文字データに変換する。
1を'1'に変換するには、1+'0'の計算を行う。

# 演習問題11-5-1

(1)

以下のプログラムは、領域Aの中に格納されたデータを16進文字列として出力するプログラムです(BINHEX.casl)。プログラムBINDECと基本的な考え方はまったく同じです。16進数なので、重みは$16^k$(k=0,1,2,3)となります。また、16進数の各桁の値は0～15になります。そこで、その値が0～9のときは、その値に'0'を加えて文字コードに変換します。10～15のときは、この値に文字コード'7'を加えると'A'～'F'になります。

```
BINHEX    START
          XOR     GR1,GR1
          LAD     GR2,1
          LAD     GR3,4            GR3:WTABLEのインデックス
          LD      GR0,A
          [ 1 ]   MINUS
                ; 重み 4096,256,16で割った数値を文字に変えてDIGITにセット
CNVLOOP   SUBA    GR3,GR2
          JMI     EXITLOOP
          LD      GR4,WTABLE,[ 2 ]      GR4←重み
                 ; GR0をGR4で割った値(GR5)と余り(GR0)を求める
          [      3      ]
DIVLOOP   CPA     GR0,GR4
          JMI     DIVFIN           GR0からGR4が引けなくなったら割り算終わり
          SUBA    GR0,GR4
          ADDA    GR5,GR2
          JUMP    DIVLOOP
DIVFIN    CPA     GR5,=[ 4 ]
          JPL     HEXDIGIT
          ADDA    GR5,='0'         文字コード'0'を加えてGR5を文字コード化する
SETDIGIT  ST      GR5,DIGIT,GR1
          [ 5 ]   GR1,GR2
          JUMP    CNVLOOP
MINUS     LD      GR7,CMINUS
          ST      GR7,DIGIT, GR1
          ADDA    GR1,GR2
          XOR     GR0,=#[ 6 ]
          ADDA    GR0,GR2
          JUMP    CNVLOOP
HEXDIGIT  ADDA    GR5,='7'
          JUMP    [   7   ]
EXITLOOP  ST      GR1,OUTLEN
          OUT     DIGIT,OUTLEN
          RET
A         DC      32000
WTABLE    DC      [   8   ]
DIGIT     DS      6
OUTLEN    DS      1
CMINUS    DC      '-'
          END
```

(2)

プログラムBINHEXにおいて、なぜ、10～15の値に'7'を加えると'A'～'F'になるのですか。

### 11.5.2 入力データ(10進の入力データ) → 内部データ(2進データ)変換

**例題 11-8**

IN命令で入力された数値データを、2進数の形式に変換するプログラム(DECBIN.casl)を作成してください。ただし、負のデータはないものとします。

キーボードから12345と打ち込んだデータをIN命令で入力すると、次のように文字コードで入力されます。

| 31 | 32 | 33 | 34 | 35 | ← 16進表記 |
|---|---|---|---|---|---|
| '1' | '2' | '3' | '4' | '5' | |

これを次のビットパターンに変えるわけです。

0011000000111001 ＝12345

このプログラムは次のようにして文字のデータを2進データに変換していきます。GR3に答えを格納することにしましょう。これには最初ゼロを入れておきます。

GR3 ← 0　　　　GR3 [0]

一番上の桁を取り出し、そこから'0'を引きます。この場合'1'なので、'1'の「文字コード－'0'」の演算結果は1ですからGR3は1になります。

GR3 [1]

次に、次の桁の文字は'2'なので2に変換して次の計算を行います。

$GR3 \leftarrow 10 \times GR3 + 2 = 10 \times 1 + 2 = 12 \quad (=1100_{(2)})$　　　　GR3 [12]

GR3の値は12になりました。
次の桁の文字は'3'なので3に変換して次の計算を行います。

$GR3 \leftarrow 10 \times GR3 + 3 = 10 \times 12 + 3 = 120 + 3 = 123 \ (=1111011_{(2)})$　　　　GR3 [123]

以下、同様にして、12345という数値を求めます。

$GR3 \leftarrow 10 \times GR3 + 4 = 10 \times 123 + 4 = 1230 + 4 = 1234 \ (=10011010010_{(2)})$　　　　GR3 [1234]

$GR3 \leftarrow 10 \times GR3 + 5 = 10 \times 1234 + 5 = 12340 + 5 = 12345 (=11000000111001_{(2)})$　　　　GR3 [12345]

この流れのプログラムを以下に示します。

```
DECBIN  START
        IN      INAREA,INLEN
        LD      GR1,INLEN
        LAD     GR2,1           1:計算用定数
        XOR     GR3,GR3         答
        XOR     GR4,GR4         INDEX
LOOP    LD      GR5,INAREA,GR4
        SUBA    GR5,='0'
;               GR3を10倍してGR5を加える
        LD      GR6,GR3
        SLA     GR3,3           GR3←GR3×8
        ADDA    GR3,GR6         GR3←GR3×9
        ADDA    GR3,GR6         GR3←GR3×10
        ADDA    GR3,GR5         GR3←GR3×10+GR5
        ADDA    GR4,GR2         カウンタ更新
        CPA     GR4,GR1         カウンタ比較
        JMI     LOOP            カウンタが小さければループ継続
        ST      GR3,A
        RET
INAREA  DS      255
INLEN   DS      1
A       DS      1
        END
```

# 演習問題11-5-2

以下のプログラムは、16進数値を入力して、それを符号無しの2進数値に変換するプログラムです(HEXBIN.casl)。たとえば、キーボードからCAFEと入力すると51966という数値(=1100101011111110(2))に変換して領域Hに格納します。

```
HEXBIN     START
           IN      INAREA,INLEN
           LD      GR1,INLEN
           LAD     GR2,1          1:計算用定数
           XOR     GR3,GR3        答
           XOR     GR4,GR4        INDEX
LOOP       LD      GR5,INAREA,GR4
           CPA     GR5,='9'
           [ 1 ]   HEXDIGIT       16進文字('A'～'F')ならば
           SUBA    GR5,='0'
;          GR3を16倍してGR5を加える
CONT       LD      GR6,GR3
           SLL     GR3,[ 2 ]      GR4←GR4×16
           ADDL    GR3,GR5        GR4←GR4×16+GR5
           ADDA    GR4,GR2        カウンタ更新
           CPA     GR4,GR1        カウンタが小さければ
           JMI     LOOP           ループを継続
           ST      GR3,H
           RET
HEXDIGIT   SUBA    GR5,=[ 3 ]     16進文字を10～15にする
           ADDA    GR5,=10
           JUMP    CONT
INAREA     DS      255
INLEN      DS      1
H          DS      1
           END
```

(1) 上記のプログラムの空欄に適当な字句を入れてください。

(2) 16倍するのにSLL命令を使っています。SLAではいけませんか。

# 総合問題11

**[演習11-1]** 次のデータの小文字の部分をすべて大文字にして出力するプログラムを作ってください (TOCHAR.casl)。データの終わりには0があるとします。

```
MESSAGE  DC   'This is a long long message'
         DC   0
```

**[演習11-2]** 次のデータから−58を探すプログラム(BINSRCH.casl)を二分探索で作成してください。

```
WTABLE  DC  -1000,-233,-58,-34,-12,0,12,122,125,456,3021,4404,9800
```

**[演習11-3]** キーボードから2つのデータを入力してその和を表示するプログラム(KEYSUM.casl)を作成してください。たとえば、'123'と'567'を入力すると、123+567を計算して690を求め、'690'と表示します。

**[演習11-4]** プログラムBINDECを改良して、たとえば−235という数値の場合、−00235という表示ではなく、先頭のゼロを抑制して−235と表示するプログラムを作成して下さい(BINDECR.casl)。

*Memorandum*

# 第12章 サブルーチン

第6章でも軽く触れたように、サブルーチンとはプログラムの部品のようなものです。本格的なプログラミングでは、全体の複雑度を下げるために、プログラムを複数のサブルーチンに分けて記述する必要があります。つまり、サブルーチンの利用は、本格的なプログラミングのための必須の技術であるといえます。逆に言えば、サブルーチンをうまく使うことによって、すっきりした分かりやすいプログラムを作ることができるのです。情報処理技術者試験の問題で出題されるプログラムの中でもサブルーチンはかなり使われています。

本章では、基本的なサブルーチンの使い方をさまざまな例を用いて説明し、次にその応用について学習します。

## 12.1 サブルーチンとは

### 12.1.1 サブルーチンの流れ(復習)

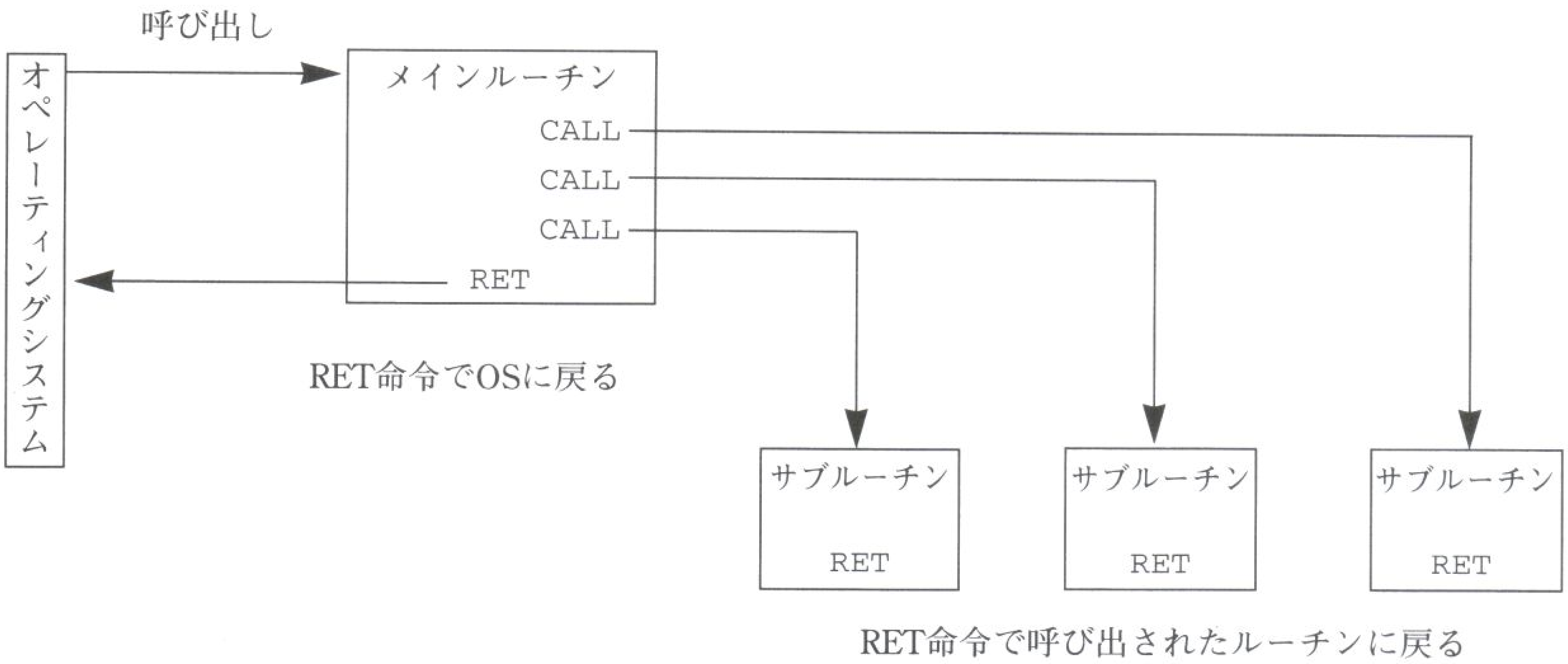
本格的なプログラムは、プログラムをメインルーチンとサブルーチンに分けて記述するのが一般的です。**メインルーチン**(main routine)とは、オペレーティングシステムによって呼び出されるプログラムであり、RET命令を実行することでオペレーティングシステムに戻ります(*1)。**サブルーチン**(subroutine)とは、プログラムの中でCALL命令によって呼び出されたプログラムのことであり、サブルーチンの中のRET命令を実行することによって、そのサブルーチンを呼び出したCALL命令の次の機械語命令に実行の制御を移します。この動作を「呼び出し側に戻る」といいます。なお、メインルーチン、サブルーチンを総称して**ルーチン**(routine)と呼びます。

**ポイント**

メインルーチンとは、オペレーティングシステムによって呼び出されたプログラム。
サブルーチンとは、CALL命令によって呼び出されたプログラム。

プログラムの実行は、必ずメインルーチンの先頭(あるいはSTART命令の開始番地に記述された番地)の機械語から始まり、実行中にCALL命令があれば、それによって呼び出されたサブルーチンの内部の機械語が実行されます。

(*1) オペレーティングシステムに戻るということは，そのプログラムの実行が終了することを意味します。

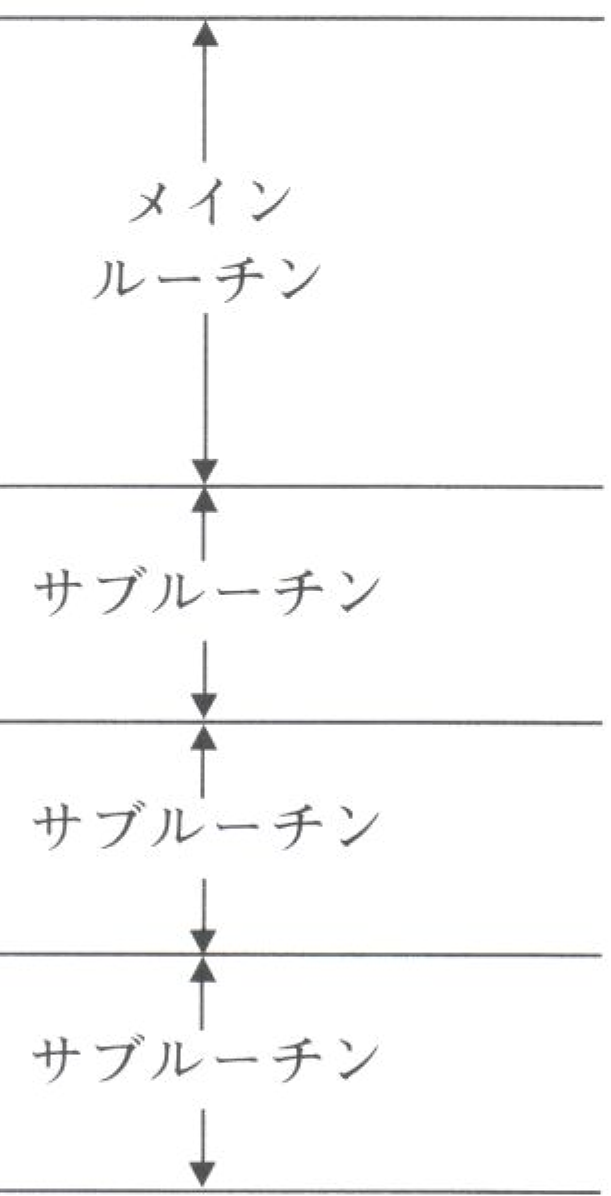



もちろん、サブルーチンを1つも含まないメインルーチンだけのプログラムであっても、プログラムとして立派に動作します。今までのプログラムはすべてメインルーチンだけで構成されていました。

サブルーチンの具体的な動作原理については、6.8節に示してあります(*1)。ここでは、復習を兼ねて、サブルーチンのより高度な動作原理について説明しましょう。

次のプログラムは、サブルーチンSUB1をメインルーチンから2回呼び出し、次にサブルーチンSUB2を呼び出し、最後にもう一度SUB1を呼び出している例です(SUBSAMP1.casl)。

```
SUBSAMP1 START
;
         CALL SUB1
         CALL SUB1
         CALL SUB2
         CALL SUB1
         RET
;
SUB1     NOP
         RET
;
SUB2     NOP
         RET
;
SUB3     NOP
         RET
;
         END
```




しかし、この例では、サブルーチンSUB3は1度も呼び出されていません。したがって、この場合には、サブルーチンSUB3の中の命令は1つも実行されません。この例からわかるように、サブルーチンは、プログラム中に記述されているからといって必ず実行されるとは限りません。サブルーチンは、

(*1) サブルーチンは、スタック上に実行の戻り番地を記憶することによって、サブルーチン側から呼び出されたルーチン側に、実行の制御を戻します。本章では、至る所に復習を設けていますが、基本的にはサブルーチンとスタックとの関係を理解していることを前提としています。もし、この辺の理屈があいまいな人は、6.8節をもう一度学習してください。

呼ばれて初めて実行されます。

## ポイント

サブルーチンは呼ばれて初めて実行される。呼ばれないサブルーチンは実行されない。

サブルーチンの中のRET命令は、そのサブルーチンを呼び出したCALL命令の次の命令に移る(リターンする)命令でしたね(復習)。前述のプログラム(SUBSAMP1)プログラムの実行をトレースしてみましょう。

ところで、サブルーチンを呼び出したCALL命令の次の機械語命令が格納されているアドレスをそのサブルーチンの**戻り番地**と呼びます。したがって、RET命令はその戻り番地に実行の制御を移す命令であると考えることができます。

```
  PR Routine Instruction      SP   OSZ   GR0  ...  GR7   Explanation
0000 MAIN    CALL 0009       752F  ---   8029 ...  F2AE  CALL 0009
0009 SUB1    NOP             752F  ---   8029 ...  F2AE
000A SUB1    RET             7530  ---   8029 ...  F2AE  Return to 0002
0002 MAIN    CALL 0009       752F  ---   8029 ...  F2AE  CALL 0009
0009 SUB1    NOP             752F  ---   8029 ...  F2AE
000A SUB1    RET             7530  ---   8029 ...  F2AE  Return to 0004
0004 MAIN    CALL 000B       752F  ---   8029 ...  F2AE  CALL 000B
000B SUB2    NOP             752F  ---   8029 ...  F2AE
000C SUB2    RET             7530  ---   8029 ...  F2AE  Return to 0006
0006 MAIN    CALL 0009       752F  ---   8029 ...  F2AE  CALL 0009
0009 SUB1    NOP             752F  ---   8029 ...  F2AE
000A SUB1    RET             7530  ---   8029 ...  F2AE  Return to 0008
0008 MAIN    RET             7530  ---   8029 ...  F2AE  Return to OS
```

上記のトレース結果の「Routine」の部分は、どのルーチンの命令を実行しているかを示したものです(*1)。また、スタックを使っているのでスタックポインタ(SP)が変化しているのが分かります。

## 演習問題12-1-1

(1) プログラムSUBSAMP1のSUB3は一度も実行されていません。一度も実行されないプログラムが含まれているこのようなプログラムはあまり望ましいものとはいえません。なぜですか。

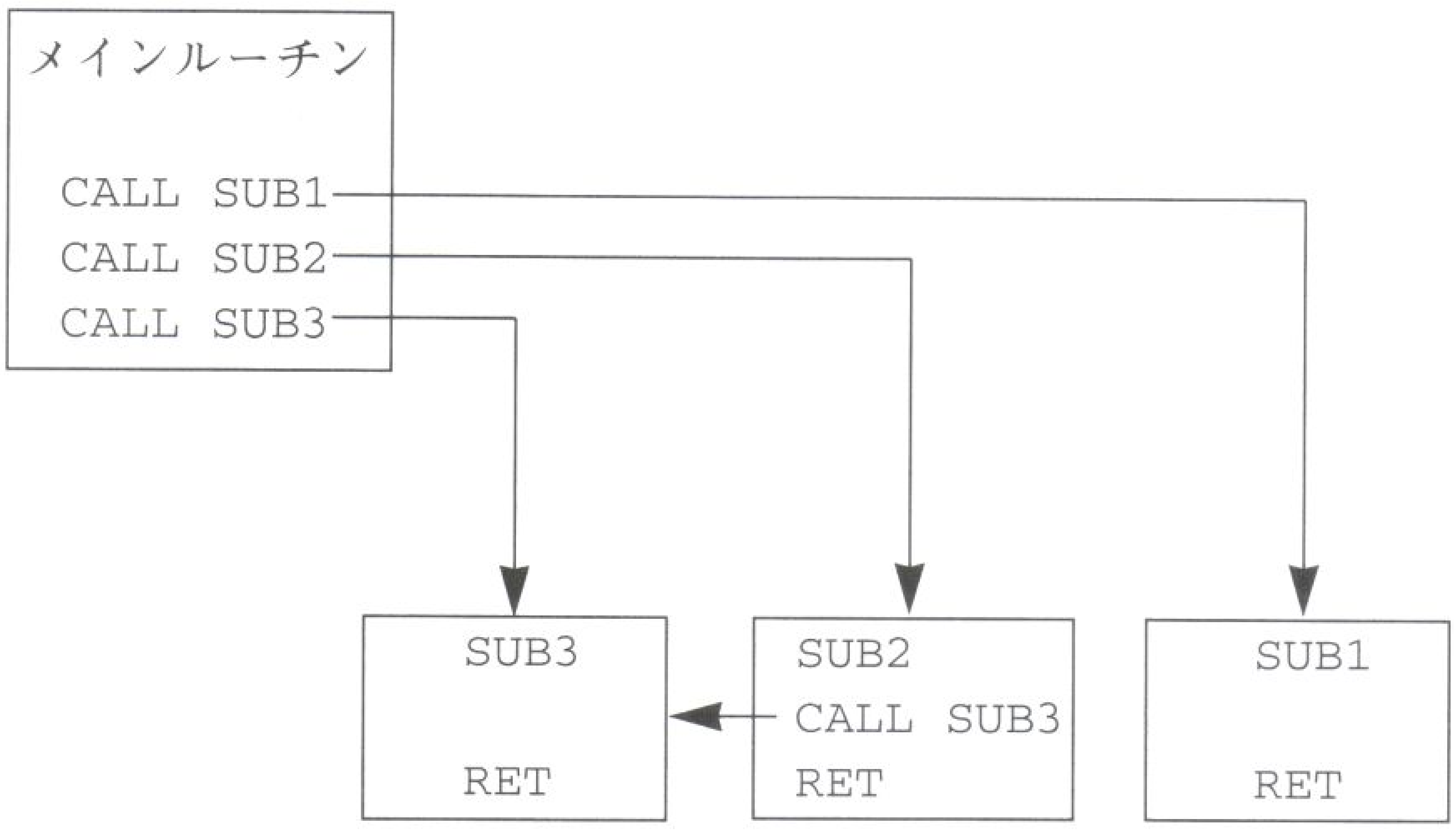
(2) サブルーチンの中には、状況によって実行されたり実行されなかったりするものがあります。どのようにすればこのようなプログラムを組めますか。

(＊1) この「Routine」を表示する機能は、残念ながら本書に付属のCASLDVには入っていません。この表示は、分かりやすいように編集しなおしたものです。

### 12.1.2 サブルーチンの中からサブルーチンを呼び出す

次の例を見てください。これは、メインルーチンから、SUB1、SUB2というサブルーチンを呼び、さらにSUB2の中からSUB3というサブルーチンを呼び出している例です。この例のように、サブルーチンの中から、またさらにサブルーチンを呼び出すことができます。

```
SUBSAMP2 START
         CALL SUB1
         CALL SUB2
         CALL SUB3
         RET
SUB1     NOP
         RET
SUB2     NOP
         CALL SUB3
         RET
SUB3     NOP
         RET
         END
```



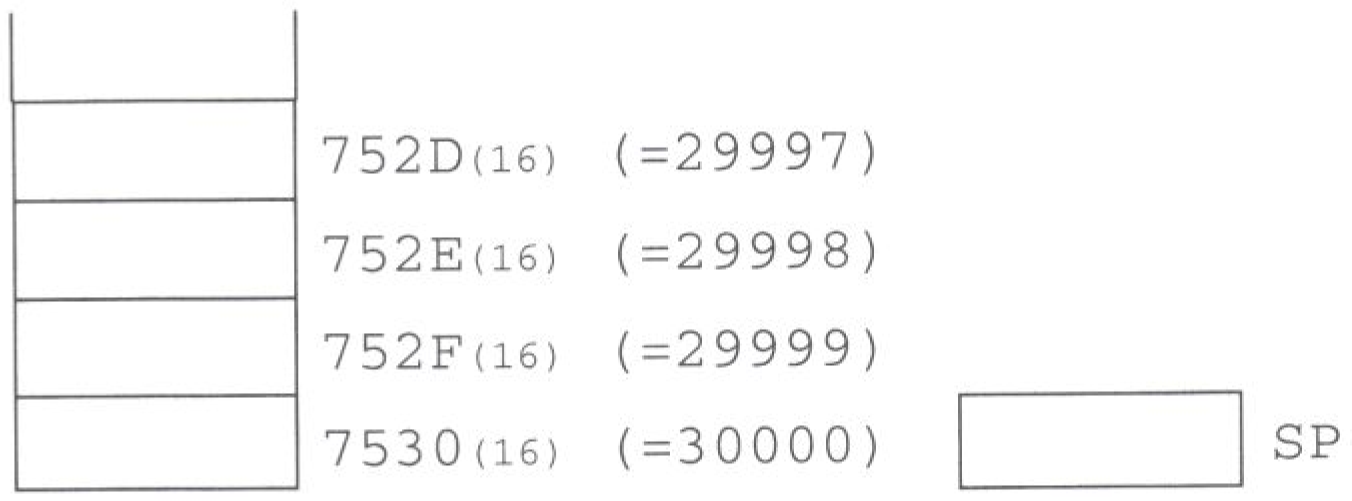

サブルーチンSUB3のRET命令は、実に巧妙な動きをします。つまり、メインルーチンから呼ばれた時にはメインルーチンに戻り、サブルーチンSUB2から呼ばれた時にはそのサブルーチンに戻ります。この仕組みは、戻り番地をCALL命令実行時にスタック上に保持(プッシュ)することによって実現しているわけですね(復習)。

なお、本書に付属のCASLDVを使って、サンプルプログラムSUBSAMP2.caslを実行するとその様子がグラフィカルに示されるので、ぜひ実行してみてください。なお、SUBSAMP2.caslはCD-ROMのSourceフォルダの中のchap12フォルダの中に入っています。

# 演習問題12-1-2

(1) 次のプログラムの「←＊」という印のついた命令を実行したときのスタックの内容を図中に記述してください。ただし、プログラム実行に先立ち、SPの値は7530(16) (=30000)にセットされているとします。

```
STKDISP START
        CALL S1
        RET
S1      PUSH 3
        CALL S2
        POP  GR1
        RET
S2      NOP      ←—— *
        RET
        END
```




なお、上記の各命令のアドレスは次の通りです。プログラムはゼロ番地からロードされたと考えて下さい。

```
  Machine     Source List (STKDISP.casl)
Adr.|Code | Line|    Program
    |     |    1| STKDISP  START
   0|8000 |    2|          CALL S1
   1|0003 |     |
   2|8100 |    3|          RET
   3|7000 |    4| S1       PUSH 3
   4|0003 |     |
   5|8000 |    5|          CALL S2
   6|0009 |     |
   7|7110 |    6|          POP  GR1
   8|8100 |    7|          RET
   9|0000 |    8| S2       NOP  ←—— *
   A|8100 |    9|          RET
    |     |   10|          END
```

(2) 本文中で、「サブルーチンの中からまたサブルーチンが呼び出せる」という記述がありました。しかし、物事には限りがあります。サブルーチン呼び出しもある程度の深さのレベルまで呼び出すことができますが、実際にはある所で呼び出せなくなります(そのときにエラーが出るとは限りませんが)。それでは何がサブルーチン呼び出しの制限となるのでしょうか(やや難)。

### 12.1.3 サブルーチンのエントリポイント

次のプログラムを見てください(SUBSAMP3.casl)。メインルーチンで、CG012345、CG23435、CG45と3回CALL命令を実行しているので、あたかも3つのサブルーチンを呼び出しているように見えます。

```
SUBSAMP3 START
         CALL  CG012345   レジスタ GR0, GR1, GR2, GR3, GR4, GR5 をゼロクリアする
         CALL  CG2345     レジスタ GR2, GR3, GR4, GR5 をゼロクリアする
         CALL  CG45       レジスタ GR4, GR5 をゼロクリアする
         RET
```

それでは、サブルーチンを含めた実際のプログラムを紹介しましょう。

```
SUBSAMP3 START
         CALL  CG012345   (1)を実行：レジスタ GR0,GR1,GR2,GR3,GR4,GR5をゼロクリアする
         CALL  CG2345     (2)を実行：レジスタ GR2, GR3, GR4, GR5 をゼロクリアする
         CALL  CG45       (3)を実行：レジスタ GR4, GR5 をゼロクリアする
         RET
CG012345 XOR  GR0,GR0   ─┬──────────
         XOR  GR1,GR1    │
CG2345   XOR  GR2,GR2   (1)   ──┬──
         XOR  GR3,GR3    │      │
CG45     XOR  GR4,GR4    │     (2)   ─┬──
         XOR  GR5,GR5    │      │    (3)
         RET  ◀──────────┴──────┴─────┴──   唯一のRET命令
         END
```

このプログラムには、サブルーチンの部分にRET命令が一つしかありません。つまり、このプログラムでは、ラベルCG2345、CG45は、サブルーチンの名前というよりもサブルーチンの途中からの入り口の名前、いわゆる**エントリポイント**と考えることができます。

CG012345の場合には、レジスタGR0のゼロクリアからレジスタGR5のゼロクリアまでが実行され、RET命令で呼び出し側に戻ります。CG2345の場合には、レジスタGR2のゼロクリアからレジスタGR5のゼロクリアまでが実行され、RET命令で呼び出し側に戻ります。CG45の場合には、レジスタGR4のゼロクリアからレジスタGR5のゼロクリアまでが実行され、RET命令で呼び出し側に戻ります。

この例のように、サブルーチンの内側のラベルをCALLして、サブルーチンを部分的に実行することもあります。

## 演習問題12-1-3

(1) 次のプログラムはサブルーチンSUB1をJUMP命令で呼び出しています。このプログラムを実行すると何が起こるでしょうか。

ヒント：サブルーチンの実行メカニズム(スタックとの関係)を思い出してください。

```
SUBSAMP4 START
         CALL SUB1
         JUMP SUB1 ←── JUMP命令で呼び出している
         RET
SUB1     NOP
         RET
         END
```

(2) 次のプログラムは、サブルーチンの中でスタック操作命令(PUSHとPOP)の個数が合っていません。つまり、PUSH命令はあるのですが、それに相当するPOP命令がありません。このプログラムを実行すると何が起こりますか。

```
SUBSAMP5 START
         CALL SUB1
         RET
SUB1     PUSH 0,GR1
         RET
         END
```

### 12.1.4　サブルーチンとJUMP命令の違い

本項では、特殊な例を用いて、サブルーチンとJUMP命令の違いについて説明します。本項で扱うプログラムは現実的なものではないかも知れませんが、CALL命令やJUMP命令の動作原理を深く理解することができます。

以下のプログラム(JUMPSMP1.casl)は、サブルーチンと似た動作を、無条件分岐であるJUMP命令を用いて実現しています。

```
JUMPSMP1   START
           JUMP  SUB1
BACK1      JUMP  SUB2
BACK2      RET
; サブルーチン(のつもり)
SUB1       NOP
           JUMP BACK1
; サブルーチン(のつもり)
SUB2       NOP
           JUMP BACK2
           END
```

つまり、JUMP命令で「サブルーチン(のつもり)」に移り、再びJUMP命令で元(呼び出した次の命令)に戻るのです。しかし、SUB1, SUB2がそれぞれわずか一回しか呼ばれていないことに注意してください。1回だけの呼び出しであれば、このような単純なJUMP命令でサブルーチンと同じ処理が実現できます。それでは、次の例のようにSUB1が2回呼ばれるとどうなるでしょう(JUMPSMP2.casl)。

```
JUMPSMP2  START
          JUMP  SUB1
BACK1     JUMP  SUB1
BACK2     RET
; サブルーチン(のつもり)
SUB1      NOP
          JUMP BACK1
          END
```

最初のSUB1の呼び出しでは「JUMP BACK1」で正しくBACK1に戻ります。この時点では何の問題もありません。しかし、2回目のSUB1の呼び出しでもBACK1に戻ってしまいます。すると、また、SUB1が呼ばれ、……というように無限ループに陥ってしまいます。2回目のSUB1の呼び出しではBACK2に戻らないといけないのです。

サブルーチンは、さまざまな場所から何度も呼ばれます。呼ばれる場所が異なれば、同じサブルーチンであっても戻り番地は異なります。私たちは、サブルーチンを記述する際に、このような面倒なことをまったく気にせず、呼び出すときは「CALL」命令、戻るときは「RET」命令を書くだけでよいのです。

でも、どうしても、「JUMP命令でサブルーチンと同じようなことをしてみたい」という人もいるかもしれません。そこで、次のプログラム(1), (2)を紹介しましょう。

### (1) 戻り番地をサブルーチンに渡す方法

次のプログラムでは、汎用レジスタに戻り番地を設定し、サブルーチンの中からその戻り番地に直接ジャンプするようにしています。汎用レジスタにアドレスを設定するには、LAD命令を使えばよいのでしたね。たとえば、ラベルBACK1のアドレスをGR7に設定するには、次のようにしました(*1)。

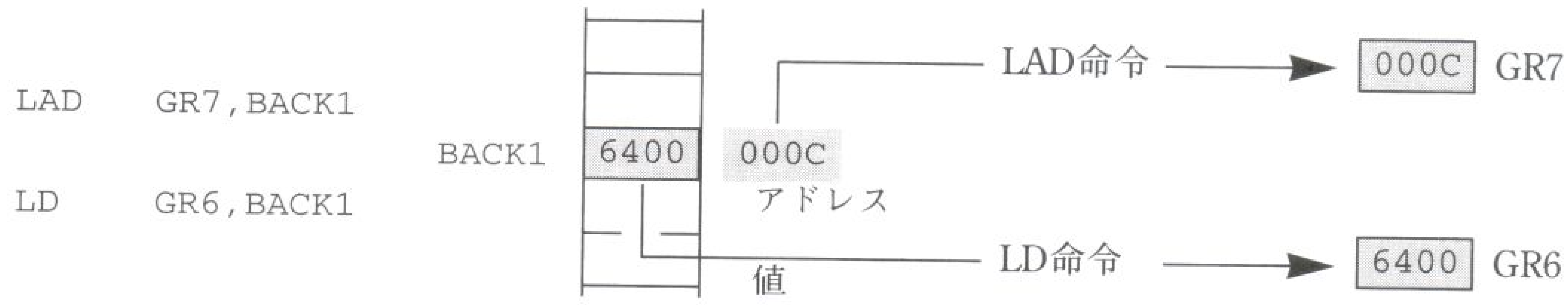
```
LAD    GR7,BACK1
```

そして、レジスタGR7に格納されたアドレスにジャンプするには、次のようにします。

```
JUMP 0,GR7
```

「0,GR7」によって生成される実効アドレスはGR7に格納された値(つまり、BACK1のアドレス)そ

(*1) BACK1のアドレスをレジスタGR7に設定するのに、LD命令で、「LD GR7,BACK1」としてはいけません。これでは、BACK1のアドレス(000C(16)とします)の語に入っている値(この例ではJUMP命令の機械語の第1語目、つまり6400(16))がレジスタにロードされてしまいます。



のものです。ジャンプ命令は、生成された実効アドレスにジャンプします(*1)。したがって、この命令「JUMP　0,GR7」によってGR7に格納されたアドレスにジャンプします。

それでは、この方式に従ったプログラムを示します(JUMPSMP3.casl)。

```
JUMPSMP3   START
           LAD    GR7,BACK1    GR7←BACK1のアドレス
           JUMP   SUB1
BACK1      LAD    GR7,BACK2    GR7←BACK2のアドレス
           JUMP   SUB1
BACK2      RET
; サブルーチン(のつもり)
SUB1       NOP
           JUMP 0,GR7   GR7に格納されたアドレスへジャンプ
           END
```

次に示す実際のトレース結果を見てみると、正しく動いていることがわかります。ただし、BACK1のアドレスは$0004_{(16)}$、BACK2のアドレスは$0008_{(16)}$です。

実は、サブルーチンを呼び出すCALL命令は戻り番地をレジスタGR7に入れる代わりに、スタックに積んでいるのです。

```
COMET_II Simulator start at 0000
 PR  Instruction       SP  OSZ GR0  GR1  GR2  GR3  GR4  GR5  GR6  GR7  Explanation
0000 LAD  GR7,0004    7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0004 [h:0004, s:4, u:4]
0002 JUMP 0009        7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0004 jump to 0009
0009 NOP              7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0004
000A JUMP 0000,GR7    7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0004 jump to 0004
0004 LAD  GR7,0008    7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0008 [h:0008, s:8, u:8]
0006 JUMP 0009        7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0008 jump to 0009
0009 NOP              7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0008
000A JUMP 0000,GR7    7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0008 jump to 0008
0008 RET              7530 --- 8029 C823 98BE E784 CAE1 BD6C ACD6 0008 Return to OS
```

### (2) ジャンプテーブルを使う方法

戻り番地を次のジャンプテーブルJMPTABLEに入れておき、その番号(0か1)をGR7に入れてサブルーチンにジャンプします。戻る時は、ジャンプテーブルのGR7番目のアドレスにジャンプします。

```
; ジャンプテーブル
JMPTABLE   DC    BACK1 ◀── 0番目
           DC    BACK2 ◀── 1番目
```

なお、「ジャンプテーブルのGR7番目のアドレスにジャンプする」プログラムはちょっと面倒です。単純に考えると、次の命令で実行できそうです。

```
JUMP   JMPTABLE,GR7   ←うまく行かない例
```

(*1) 「JUMP BACK1」で生成される実効アドレスは「BACK1のアドレス」でした。だから、BACK1へジャンプするのでしたね(復習)。

この命令によって生成される実効アドレスは、「JMPTABLEのアドレス(000E(16)番地に配置されているとします)+GR7」です。もし、GR7に1が入っているとすると、「000E(16)+1」→「000F(16)」となり、JMPTABLEの2つ目の語のアドレスを意味します。これが実効アドレスとなりますので、なんと、JMPTABLEの2つ目のアドレスにジャンプ(実行の制御を移す)してしまいます。ここには、BACK2のアドレスが格納されているだけですから、それを機械語と思って強引に実行してしまい、多くの場合、暴走という現象を引き起こします(*1)。

ジャンプテーブル中に格納されているアドレスにジャンプするなら、一度、そのアドレスを取り出して、そこにジャンプすればよいでしょう(JUMPSMP4.casl)。なお、この内容は6.6節で学習済みです。

```
JUMPSMP4  START
          LAD   GR7,0      GR7←0:JMPTABLEの0番目を指示（実はそれはBACK1のアドレス）
          JUMP  SUB1
BACK1     LAD   GR7,1      GR7←1:JMPTABLEの1番目を指示（実はそれはBACK2のアドレス）
          JUMP  SUB1
BACK2     RET
; サブルーチン（のつもり）
SUB1      NOP
          LD    GR7,JMPTABLE,GR7   GR7←JMPTABLEのGR7番目に格納されたアドレス
          JUMP  0,GR7      そのアドレスへジャンプ
; ジャンプテーブル
JMPTABLE  DC    BACK1
          DC    BACK2
          END
```

## 演習問題12-1-4

サブルーチンの中では、スタック操作命令であるPUSH命令とPOP命令の個数は一致していなければなりません。なぜでしょうか。

### 12.1.5 プログラム単位とサブルーチン

一つのファイルの中に記述されているプログラムは、その中にいくつサブルーチンがあっても一度にアセンブル(*2)することができます。

次のプログラムがファイルSUBSAMP6.caslに格納されているとします。

(*1) このようなミスは初心者にはよくあることです。そこで、本書に付属のCASLDVは、このような現象をチェックし、利用者に伝えるようにしています。上記のプログラムを実行すると、次のようなメッセージが表示され、プログラムの実行が停止します。

警告：000C番地の命令は機械語としてアセンブルされていません。(Code=65)
　　おそらく暴走しているものと思われます。

付属のCASLDVは、ロードされた機械語の種別を認識し、ロードしていない場所やデータ領域を機械語として実行しようとするとエラーを出すようにしています。しかし、本来のコンピュータはこのようなことをしないので、CASLDVの実行時にこの機能を外すこともできます。

(*2) 「一度にアセンブルする」という意味は、一回のアセンブラの起動で一つのオブジェクトファイルを生成することを言います。

```
SUBSAMP6  START
          CALL   SUB1
          CALL   SUB2
          CALL   SUB3
          RET
SUB1      NOP
          CALL   SUB3
          RET
          END
```

SUBSAMP6.casl

それでは、これを本書に付属のアセンブラ(*1)でアセンブルしてみましょう。すると、次のようなメッセージが表示されます(*2)。

参考：未定義の外部参照名がありました。これらを定義しているサブルーチンをリンクしないと実行できません。

外部参照名　引用行
SUB2　　　　3
SUB3　　　　4, 7

これは、「参考」と書いてあるように、エラーメッセージではありません。このメッセージは、この中で3行目に引用されているSUB2と4行目と7行目に引用されているSUB3が、「このプログラムのどこにも定義されていませんよ」というメッセージです。エラーが無いので、SUBSAMP6.cometという機械語ファイルが生成されます。

このプログラムには、前述のようにエラーは含まれていません。しかし、このままでは対象となるサブルーチンがないので実行させることはできません。この機械語プログラムを本書に付属のCASLDVで強引に実行すると、次のエラーメッセージが表示されます。

....Loading ObjectFile:SUBSAMP6.comet ProgramLabel:SUBSAMP6, Start:0000-->0000
ファイル(SUBSAMP6.comet)中の外部参照名(SUB2)に対応するラベルがありませんでした。
ファイル(SUBSAMP6.comet)中の外部参照名(SUB3)に対応するラベルがありませんでした。
ローディングに失敗しました。

つまり、SUB2とSUB3のプログラムがないから実行できないというのです。実は、これらのサブルーチンは、別のファイルで次のように定義されています。それぞれ、SUB2.casl、SUB3.caslというファイルに格納されているとします(*3)。

(*1) 本書のCASLDVには、非GUI版のアセンブラが含まれております。CASLDVをインストールするとbinフォルダの中にCASL_II.exeというプログラムがコピーされます。これが、そのアセンブラです。

(*2) このメッセージは非GUI版のものです。CASLDV(GUI版)はこれとは異なる形式になります。

(*3) CASL IIのプログラムを格納するファイル名の規則は、具体的には決まっていません。しかし、ファイル名は、一般にプログラム名と同じ名前にした方が分かりやすいでしょう。なお、本書に付属のCASLDVでは、ソースプログラムのファイル名は「~.casl」、オブジェクトファイル(機械語ファイル)のファイル名は「~.comet」と拡張子を規定していますが、ファイル名は任意です。仮に、SUB2のプログラムをSSS.caslというファイルに入れてもOKです。本書に付属のCASLDVでSSS.caslをアセンブルするとSSS.cometというファイルが生成され、その中にSUB2の機械語が入ります。

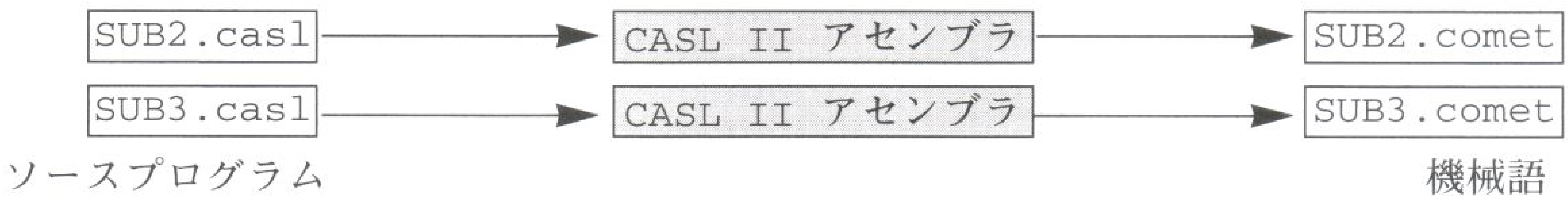
```
SUB2    START
        OUT    MES,MESLEN
        RET
MES     DC     'SUB2 EXECUTE'
MESLEN  DC     12
        END
```

SUB2.casl

```
SUB3    START
        OUT    MES,MESLEN
        RET
MES     DC     'SUB3 EXECUTE'
MESLEN  DC     12
        END
```

SUB3.casl

SUBSAMP6.caslを正しく実行させるためには、これらを別途アセンブルして機械語を生成しておく必要があります。




このプログラムの実行には、この3つの機械語ファイル(SUBSAMP6.comet, SUB2.comet, SUB3.comet)が必要になります。

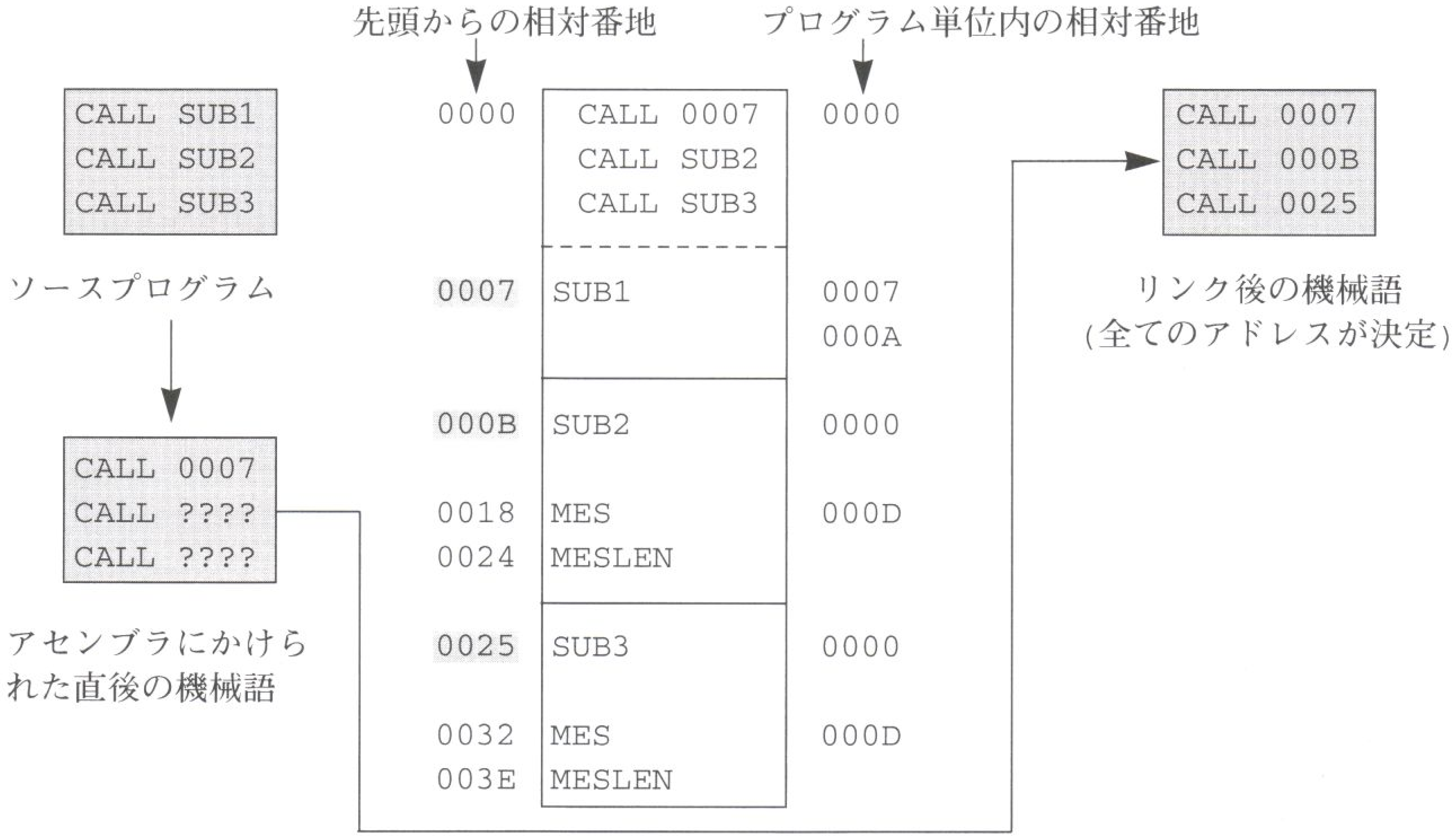
ところで、このように一回のアセンブラの処理の単位を**プログラム単位**(program unit)と呼びます。CASL IIの場合には、一つのファイルに格納されたSTART命令からEND命令までのプログラムがプログラム単位です。

著者が開発した処理系の実行の方法は、コマンドラインの方法(*1)とGUIの方法(CASLDV)で、多少異なります。しかし、どちらの方法でも、メインルーチンを一番最初に指定します。コマンドライン版ならば、

```
comet_II SUBSAMP6.comet SUB2.comet SUB3.comet -t
```

と指定します。最後の−tは実行をトレースすることを意味します。これによって、COMET IIシミュレータが、この3つの機械語のラベルを結びつけて一つの実行可能な機械語にまとめます。具体的には、次ページの図に示すように3つの機械語プログラムを順番に配置して行きながら、各ラベルのアドレスを決めていきます。たとえば、この例では、まず、指定されたアドレスからメインルーチンSUBSAMP6を配置します。SUBSAMP6を配置中にラベルSUB1を持つ機械語が配置されます。この時点で、SUB1のアドレスを決定することができます。この例では、SUB1のアドレスは先頭からの相対番地で$0007_{(16)}$となります。次に、メインルーチンSUBSAMP6の配置が完了したら、その後ろにサブルーチンSUB2の機械語を配置します。SUBSAMP6の最後の機械語のアドレスが$000A_{(16)}$ですので、SUB2は$000B_{(16)}$番地目から配置します。これが、サブルーチンSUB2の先頭アドレスになります。SUB2のMESは$0018_{(16)}$番地、MESLENは$0024_{(16)}$番地になります。SUB2を配置し終わったら、次にサブルーチンSUB3を配置します。これは$0025_{(16)}$番地から配置されます。この番地が、サブルーチンSUB3の先頭アドレスになります。SUB3のMESは$0032_{(16)}$番地、MESLENは$003E_{(16)}$番地になります。これで、未解決名(アドレスの不明なラベル)であるSUB2, SUB3のアドレスが決定されました。未決定であったメインルーチンのCALL命令のアドレスを決定することができました。これで、CALL命令を完全に機械語に変換することができ、実行可能になりました。この処理を**リンク**(link)処理と呼びます。

(*1) コマンドライン版(comet_II)は、本書に付属のCD-ROMには含まれておりません。コマンドライン版の入手方法は序文をご覧下さい。




ただし、CASL IIアセンブラ言語の場合、このように結び付けられる対象のラベルは、あくまでもSTART命令に付けられたプログラムの名前だけです。つまり、一つのプログラム単位の中で定義されたラベル(START命令に付けられたもの以外)は、リンクの対象になりません。したがって，そのようなラベル(上記の例では、MESやMESLEN)は同一のプログラム単位の中だけでしか使えませんので、注意してください。逆に言えば、SUB2とSUB3で共に同じラベル名MES, MESLENを使っていても、それは異なるものとして扱われるので、まったく問題ありません。

ちなみに、このプログラムを動作させてみると、つぎのような表示が得られます。

```
SUB3 EXECUTE
SUB2 EXECUTE
SUB3 EXECUTE
```

### 12.1.6 メインルーチンとサブルーチンの区別

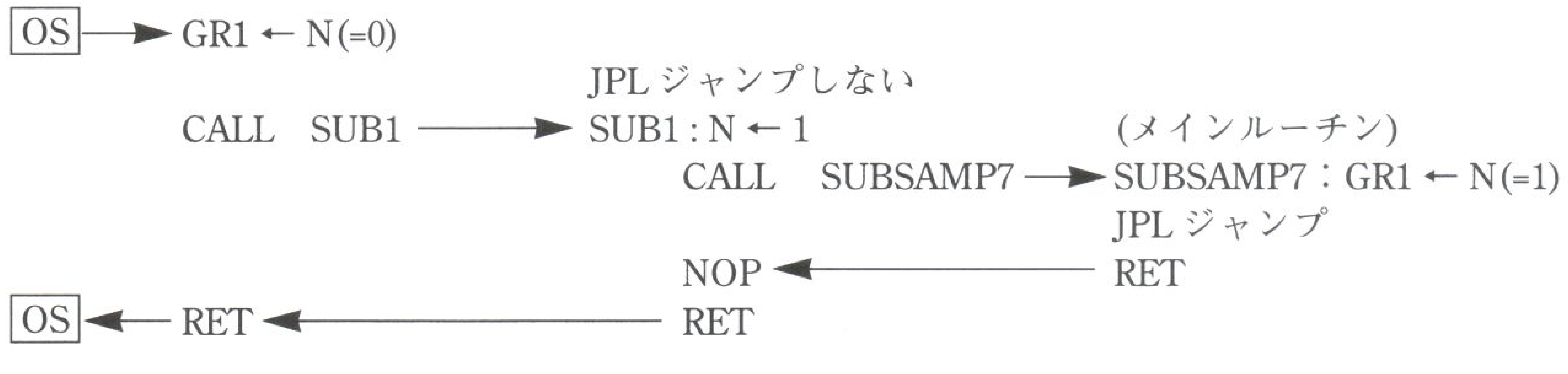
実際には、メインルーチンとサブルーチンの区別というものはありません。12.1.1項で述べたように、オペレーティングシステムから呼ばれた場合(すなわち、一番最初に実行されるプログラム)がメインルーチンで、メインルーチンあるいはサブルーチンから呼ばれるプログラムがサブルーチンになります。

ちょっとトリッキーですが、次のプログラムは、サブルーチンSUB1からメインルーチンSUBSAMP7を呼び出している例です(SUMSAMP7.casl)。

```
SUBSAMP7  START
; メインルーチン
          LD    GR1,N
          JPL   FIN
          CALL  SUB1
FIN       RET   ;サブルーチンとして呼ばれたときは呼び出し側へ、OSから呼ばれたときはOSへ帰る
N         DC    0
; サブルーチン
SUB1      LAD   GR1,1
          ST    GR1,N
          CALL  SUBSAMP7 メインルーチンを呼び出す
          NOP
          RET
          END
```

このプログラムの動作を追ってみましょう。プログラムの実行はメインルーチンから始まります。最初に、Nの値であるゼロがGR1にロードされます。Nは「DC 0」によって初期値0になっているからです。さて、「LD GR1,N」によってGR1には0がロードされますから、ZF(Zero Flag)が立ちます。したがって、JPL命令では分岐しません。そのまま、次の命令である「CALL SUB1」を実行し、サブルーチンSUB1に制御が移ります。この中で、Nの値を1に変更しています。そして、今度は、メインルーチンをサブルーチンとして呼んでいます(CALL　SUBSAMP7)。そして、実行は、SUBSAMP7の最初の命令「LD GR1,N」になり、今度はGR1に1がロードされます。すると、次の分岐命令JPLの条件が成立し、ラベルFINにジャンプします。つまり、CALL SUB1をスキップして実行しないのです(もし、実行するとSUB1の中で再びメインルーチンが呼ばれ、結局、無限ループになってしまいます)。FINには、RET命令があります。サブルーチンとして呼ばれているので、この場合、呼び出されたCALL命令の次の命令に移ります。それは、SUB1のNOPですね。NOP命令は実行しても何も変化しません。次のRET命令を実行すると、今度はメインルーチンに戻り、メインルーチンのRET命令を実行します。このとき、メインルーチンはOSから呼ばれているので、今度はOSに戻ります。

このプログラムの実行の概略を以下に示します。




なお、本書で提供しているCASLDVでは、OSに戻るRETだけは特別な処理をしているため、そのRETを実行してもスタックポインタは変化しませんので、ご注意下さい。

## 12.2 サブルーチン間でのデータの渡し方

サブルーチンを利用する際に、呼び出し側のデータをサブルーチンに渡すことが必要になります。また、サブルーチン側で計算した結果を呼び出し側に戻すことも必要です。

たとえば、C←A×Bという乗算を行うサブルーチンMULTがあるとします。メインルーチンからMULTを呼び出すときに、掛ける2つの値(A,B)を渡して、結果(C)をMULTから返してもらわなければなりません。これを実現する一番簡単な方法は、汎用レジスタを使う方法です。たとえば、「サブルーチンMULTはレジスタGR1とGR2の内容を乗算し、その結果をレジスタGR0に返す」と約束しておけば、次のようにして、C←A×Bを求めることができます。

```
        LD    GR1,A
        LD    GR2,B
        CALL  MULT
        ST    GR0,C
           :
  A     DC    24
  B     DC    33
  C     DS    1
```

しかし、これだけでは何でサブルーチンを使うのかピンと来ないかもしれません。この例では、24×33の計算しかしていないので、このデータをわざわざMULTに渡す意味があまりに明確に感じられません。そこで次の例を見てください。これは、X×Y+A×Bを計算するプログラムです。

```
        LD    GR1,A
        LD    GR2,B
        CALL  MULT
        LD    GR7,GR0       GR7←A×B
        LD    GR1,X
        LD    GR2,Y
        CALL  MULT
        ADDA  GR0,GR7       GR0←X×Y+GR7=(X×Y)+(A×B)
           :
  A     DC    24
  B     DC    33
  X     DC    12
  Y     DC    56
```

この中では、異なるパラメータ(24,33と12,56)をMULTに渡すことで2つの乗算を実現しています。もし、これをサブルーチンを使わないで作ると、乗算プログラムを2つ作らねばならず、効率が悪くなります。

本節では、このようなサブルーチンへのデータの授受について説明します。データの授受については、レジスタの他にさまざまなものがあります。この方法は、また、同一のプログラム単位内のサブルーチンと他のプログラム単位内のサブルーチンで異なります。

表：12-1　サブルーチン間のデータの受け渡し方

| プログラム単位 | レジスタ | 共有ラベル | アドレス渡し | スタック |
|---|---|---|---|---|
| 同一のプログラム単位 | ◎ | ◎ | ○ | △ |
| 異なるプログラム単位 | ◎ | ☆ | ○ | △ |
| 参照個所 | 12.2.1 | 12.2.2 | 12.2.3 | 12.2.4 |

◎：問題なく可能
○：アドレスをレジスタで渡せば可能
△：かなりトリッキーな方法を使えば可能
☆：ラベルのアドレスをサブルーチンに渡してアクセスすれば できるが面倒

### 12.2.1　レジスタ渡し

汎用レジスタは、すべてのルーチンで共通に使えます。したがって、同一のプログラム単位でも異なるプログラム単位でも同じようにデータの受け渡しが可能です。次の例は、実際に、A,Bをレジスタ渡しでサブルーチンMUTLREGに渡して、計算結果A×BをGR0に受け取る例です。

```
MULTS1 START
       LD   GR1,A
       LD   GR2,B
       CALL MULTREG
       ST   GR0,C
       RET
A      DC   24
B      DC   33
C      DS   1
       END
```

MULTS1.casl

```
MULTREG  START
         XOR   GR0,GR0     積を格納するところ
         CPA   GR2,=0      もし、Bが
         JZE   FINISH      0なら答えはゼロ
         LAD   GR3,1       1,定数
MLTLOOP  ADDA  GR0,GR1     GR0←GR0+GR1, 積を求める
         SUBA  GR2,GR3     カウントダウン
         JPL   MLTLOOP     カウンタが1以上ならMLTLOOPへ
FINISH   RET
         END
```

MULTREG.casl

## 演習問題12-2-1

サブルーチンMULTREGは、その中でGR3の値を1にセットして使っています。MULTREGを呼び出しても、GR3の値が変更されないように、MULTREGを書き直しました。空欄に適当な命令を入れてください。

```
MULTREG START
        [   1   ]
        XOR  GR0,GR0
        CPA  GR2,=0    もし、Bが
        JZE  FINISH    0なら答えはゼロ
        LAD  GR3,1     1,定数
MLTLOOP ADDA GR0,GR1   GR0←GR0+GR1, 積を求める
        SUBA GR2,GR3   カウントダウン
        JPL  MLTLOOP   カウンタが1以上ならMLTLOOPへ
FINISH  [   2   ]
        RET
        END
```

### 12.2.2 共有ラベル

同一のプログラム単位内のラベルは、そのプログラム内であればどこからでも見えます(*1)から、次に示すサブルーチンMULTMEM内で定義されているラベル MA、MB、MCは、メインルーチンでも参照できます(MULTS2.casl)。したがって、この領域に、A、Bの値をコピーし、サブルーチンを呼びます。サブルーチンはMC ← MA × MB を計算しますので、MCの値をCにコピーします。

```
MULTS2  START
        LD   GR0,A      Aを
        ST   GR0,MA     MAへコピー
        LD   GR0,B      Bを
        ST   GR0,MB     MBへコピー
        CALL MULTMEM
        LD   GR0,MC     MCを
        ST   GR0,C      Cへコピー
        RET
A       DC   24
B       DC   33
C       DS   1
MULTMEM XOR  GR0,GR0    積を格納するところ
        LD   GR1,MA     GR1←MA(=A)
        LD   GR2,MB     MB(=B)をロードし、もし、MBが
        JZE  FINISH     0なら答えはゼロ
        LAD  GR3,1      1;定数
MLTLOOP ADDA GR0,GR1    GR0←GR0+GR1, 積を求める
        SUBA GR2,GR3    カウントダウン
        JPL  MLTLOOP    カウンタが1以上ならMLTLOOPへ
FINISH  ST   GR0,MC     GR0(積) → MC
        RET
MA      DS   1
MB      DS   1
MC      DS   1
        END
```

この方法は、レジスタ渡しに比べると煩雑です。しかし、レジスタの総個数は8個しかないので、8個以上のデータを渡す際には便利です。残念ながら、この方法は、同一のプログラム単位内だけで、プログラム単位が異なればできません。なぜなら、CASL IIのプログラムでは、異なるプログラム単位のラベルが参照できるのはSTART命令に付けられたプログラムの入り口名だけであるからです。

## 演習問題12-2-2

プログラムMULTS2では、レジスタGR0〜GR3を使っています。プログラムMULTS2を呼び出しても、戻ってきたときの汎用レジスタの値が元の値と同じになっているようにするには、どのようにしたらよいでしょうか。

(＊1) 「見える」という表現は、コンピュータを擬人化した表現で、そのラベルが「有効で使える」ということを意味しています。

### 12.2.3 アドレス渡し

**アドレス渡し**とは、汎用レジスタにデータが格納されている語のアドレスを渡し、サブルーチンの中で直接、その語を操作する方式です。

次のプログラムは、まず、メインルーチン側で、メインルーチンのラベルA, B, Cのアドレスを汎用レジスタに格納してサブルーチンに渡します。次に、サブルーチン側では、その汎用レジスタに格納されたアドレスの値を同じ汎用レジスタに入れて(たとえば、レジスタGR1にAのアドレスが格納されているときAの値をGR1にセットするには「LD　GR1,0,GR1」によって実現)A×Bの計算をしている例です。レジスタGR3にCのアドレスが格納されているので、それを利用して「ST　GR0,0,GR3」によってGR0に求められている解をCの領域に直接格納しています。

```
MULTS3   START
    LAD  GR1,A  GR1←Aのアドレス
    LAD  GR2,B  GR2←Bのアドレス
    LAD  GR3,C  GR3←Cのアドレス
    CALL MULTADR
    RET
A   DC   24
B   DC   33
C   DS   1
    END
```

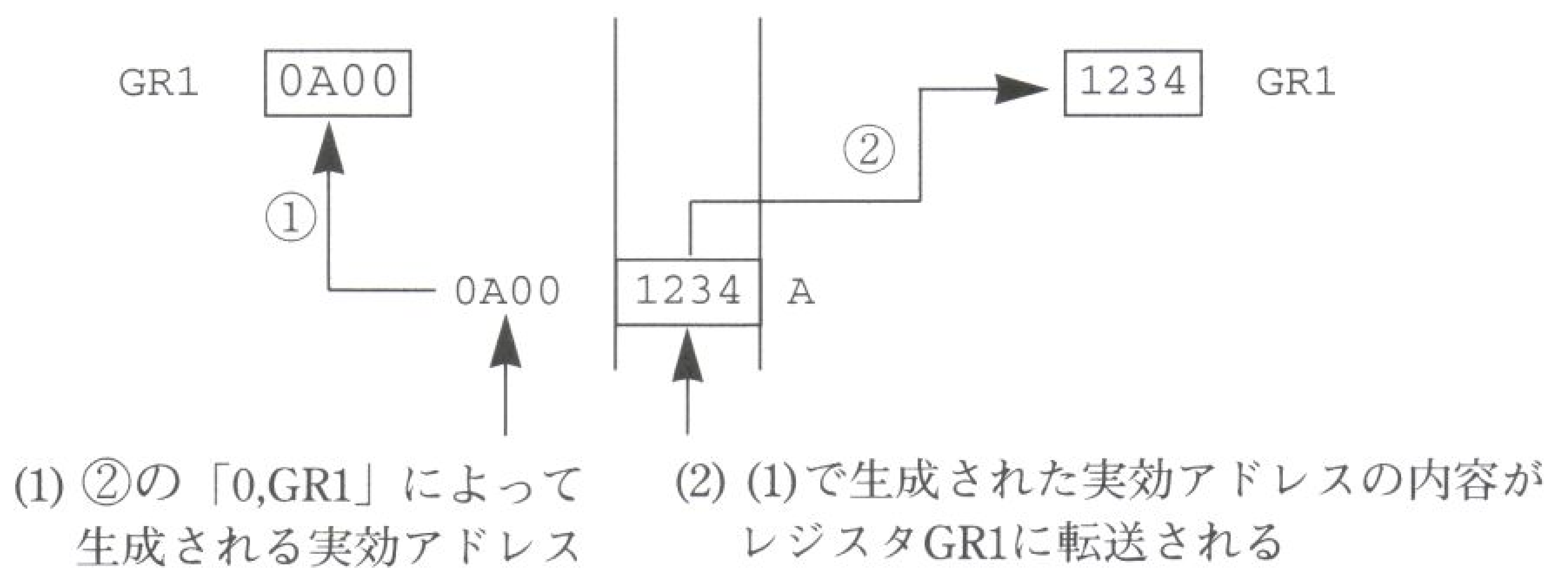
MULTS3.casl

```
MULTADR START
        XOR   GR0,GR0      積を格納するところ
        LD    GR1,0,GR1    GR1←Aの値(GR1のアドレスの値をGR1へロード)
        LD    GR2,0,GR2    GR2←Bの値をロードし、もし、Bが
        JZE                FINISH      なら答えはゼロ
        LAD   GR4,1        1;定数
MLTLOOP ADDA  GR0,GR1      GR0←GR0+GR1,積を求める
        SUBA  GR2,GR4      カウントダウン
        JPL   MLTLOOP      カウンタが1以上ならMLTLOOPへ
FINISH  ST    GR0,0,GR3    (積) → C (GR3のアドレス)
        RET
        END
```

MULTADR.casl

「LD　GR1,A」・「LD　GR1,0,GR1」という動作のメカニズムを図示すると次のようになります。

① LAD　GR1,A　　　①はAのアドレスをGR1へセットする
② LAD　GR1,0,GR1　　②はAの値をGR1にセットする




「LD GR1,0,GR1」という記述は、「0,GR1」によって生成される実効アドレス、すなわち「GR1の値＝Aのアドレス」の内容をレジスタGR1に入れるという命令です。この1命令でメインルーチンのAの値をサブルーチンの中でレジスタGR1に取り込めてしまいます。

ただ、この方法も、汎用レジスタを単純に使う方法に比べると多少やっかいです。この方法の利点を整理すると次のようになるでしょう。

(1) 異なるプログラム単位の呼び出し側の領域のデータの設定がサブルーチン側で可能

(2) テーブルのような多数のデータの先頭アドレスを渡せば、テーブルをサブルーチン内でアクセス可能

(2)の意味が多少わかりにくいかも知れませんので例を示します。次のプログラムは、テーブルのアドレスをサブルーチンに渡し、そのアドレスが指すテーブル中のすべてのデータをゼロが現れるまで加えて、レジスタGR0に入れて返すというものです。

```
SUMUPS1 START
        LAD  GR1,DTABLE GR1←DTABLEのアドレス
        CALL SUMTABLE
        ST   GR0,SUM    集計値(GR0)→SUM
        RET
DTABLE  DC   4,6,7,4,2,5,5,1,0
SUM     DS   1
        END
```

SUMUPS1.casl

```
SUMTABLE START
         XOR  GR0,GR0     集計する場所
         LAD  GR3,1       1:定数
LOOP     LD   GR2,0,GR1   テーブルから1つ取り出す
         JZE  FINISH      それがゼロなら終わり
         ADDA GR0,GR2     テーブルの値を加える
         ADDA GR1,GR3     テーブルの位置をカウントアップ
         JUMP LOOP
FINISH   RET
         END
```

SUNTABLE.casl

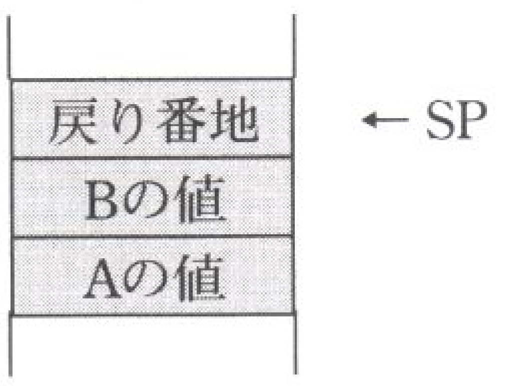
### 12.2.4 スタックを使う方法

スタックは、主記憶装置の一部なので、その場所さえ分かれば、スタックにアクセスすることができますが、COMET IIでは、スタックレジスタSPの値を他の汎用レジスタに移すことができません。そのため、スタックのアドレスを手に入れて処理するプログラムが書けません。スタック中に積まれたデータを確実に受け取る唯一の方法はPOP命令だけです。

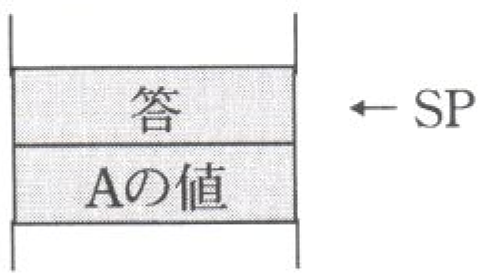
たとえば、乗算のプログラム(C←A×B)であれば、次のプログラムの最初の2行に示すようにして、スタックにデータを積み上げます。GR1にAの値、GR2にBの値があるとします。

```
     PUSH 0,GR1    Aの値をPUSH
     PUSH 0,GR2    Bの値をPUSH
     CALL MULTSTK
①→  POP  GR0      計算結果をスタックから取り出して
     ST   GR0,C    Cへコピー
②→  POP  GR0      タイミング合わせの空ポップ
```

CALL命令実行直後のスタックは次のようになっています。




サブルーチン実行後、戻ってきたときのスタックは、サブルーチンMULTSTKの実行によって、次のようになっています。



したがって、CALL命令のすぐ下の最初のPOP命令①で答えを取り出し、レジスタGR0に格納しています。その次のPOP命令②は、一つのルーチンの中で用いるPUSH命令とPOP命令の数を合わせるために使っています。

それでは、この処理の完全なプログラム例を示しましょう。最初にメインプログラム(MULTS4.casl)を示します。

```
MULTS4  START
        LD    GR1,A
        LD    GR2,B
        PUSH  0,GR1        Aの値をPUSH
        PUSH  0,GR2        Bの値をPUSH
        CALL  MULTSTK
        POP   GR0
        ST    GR0,C        Cへコピー
        POP   GR0          タイミング合わせの空ポップ
        RET
A       DC    24
B       DC    33
C       DS    1
        END
```

それでは、サブルーチンMULTSTK(MULTSTK.casl)はどのようになっているのでしょうか。それを次に示します。

```
MULTSTK START
        POP   GR7          戻り番地を受け取る
        POP   GR2          Bの値を受け取る
        POP   GR1          Aの値を受け取る
        XOR   GR4,GR4      解を格納するところ
        CPA   GR2,=0       MB(=B)をロードし、もし、MBが
        JZE   FINISH       0なら答えはゼロ
        LAD   GR3,1        1,定数
MLTLOOP ADDA  GR4,GR1      GR4←GR4+GR1, 積を求める
        SUBA  GR2,GR3      カウントダウン
        JPL   MLTLOOP      カウンタが1以上ならMLTLOOPへ
FINISH  PUSH  0,GR1        空PUSH
        PUSH  0,GR4        答えを戻す
        PUSH  0,GR7        戻り番地を戻す(とても重要)
        RET
        END
```

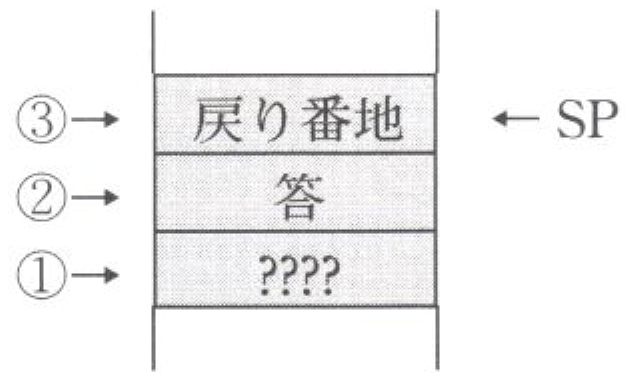
このプログラムは、実はかなりトリッキーなことをしています。まず、サブルーチンに入るとすぐに、強引に3回POP命令を実行しています。3回のPOP命令終了後の段階で、すでにスタックは正しい状態になっていません。なぜなら、戻り番地まで取り出してしまうからです。



このPOP命令実行後、スタック上にはデータはそのまま残っているものの、もはやプログラムでアクセスすることはできません。SPの値を元に戻すにはPUSHするしかなく、別のデータをPUSHするとその別の値がスタックに積み上げられてしまうからです。その意味では、スタック上のデータは無くなってしまった(もうアクセスできない)と考える方が良いでしょう。そのため、その領域を？？？？で示しています。

さて、スタック上からサブルーチンの戻り番地すら無くなってしまいました。このまま、RETすると暴走する可能性が高いでしょう。

そこで、乗算の計算が終了したら、今度は、PUSH命令を使ってスタックを次のように元に戻します。最初の部分①には、データを入れる必要がないので適当なデータでPUSHし(*1)、次に答えをPUSHし②、最後にGR7に保存していた戻り番地をPUSHします③。




スタックが完全に修復できたので、これでRET命令を実行すると呼び出し側へ戻ります。何でこんな面倒なことをするのかと思われるかもしれません。普通のサブルーチンの使い方では必要ないのですが、再帰的サブルーチンを作る際には、このようなスタック操作が必要になることがあります。

(＊1) 実際にはGR1の値をPUSHしているので、Aの値がPUSHされます。

## 12.3 部品としてのサブルーチン

前節までは、サブルーチンに関する個々の機械語命令について説明してきましたが、本節では、サブルーチンを部品として用いたプログミングについて例題を通して説明します。

> **例題**
> D ← (A×B)×Cを計算するプログラムを作りなさい。

乗算の部分をサブルーチン化することによって、メインルーチンをすっきりしたものにすることができます。プログラムMULTSAMPはA≧0,B≧0,C≧0の各値に対して乗算(C←A×B)を行うプログラムです(MULTSAMP.casl)。

```
MULTSAMP START
         LD    GR3,A       Aの値をレジスタGR3に入れてMULTSUBに渡す
         LD    GR4,B       Bの値をレジスタGR4に入れてMULTSUBに渡す
         CALL  MULTSUB     サブルーチンMULTSUBの呼び出す
         LD    GR3,GR2     答(A×B)がレジスタGR2に戻るので、次の計算のために、GR3に格納
         LD    GR4,C       Cの値をレジスタGR4に格納
         CALL  MULTSUB     (A×B)×Cを計算
         ST    GR2,D       D ← (A×B)×C
         RET
A        DC    25
B        DC    4
C        DC    10
D        DS    1
;
; サブルーチン(MULT) C ← A × B を計算する
;       入力   GR3:A
;              GR4:B
;       出力
;              GR1:オーバーフローフラグ(0:していない、1:している)
;              GR2:C
;
MULTSUB  XOR    GR1,GR1      GR1←0, オーバーフローフラグ (0:していない,1:している)
         XOR    GR2,GR2      GR2←0, 解を格納するレジスタ
         LD     GR3,GR3      フラグレジスタをGR3(Aの値)で設定する
         JZE    FINISH       もしAがゼロなら計算しない
         LD     GR4,GR4      フラグレジスタをGR4(Bの値)で設定する
         JZE    FINISH       もしBがゼロなら計算しない
         LAD    GR5,16       GR5←16, 16回ループする (GR5はカウンタ)
         LAD    GR6,1        GR6←1, 計算用定数
LOOP     SRA    GR4,1        GR4(B)のビットを押し出す
         JOV    ADDVALUE     押し出された最下位ビットが1なら加算処理へ
CONT     SLA    GR3,1        Aを1ビット左へシフト
         SUBA   GR5,GR6      カウントダウン
         JNZ    LOOP         ゼロでなければLOOPへ
FINISH   RET                 ;メインルーチンに戻る
ADDVALUE ADDA   GR2,GR3      解(GR2)に重み付けされたA(GR3)を加える
```

```
         JOV   SETFLAG    もし、オーバーフローしたならフラグをセット
         JUMP  CONT       主処理へ戻る
SETFLAG  LAD   GR1,1      オーバーフローフラグをセット
         JUMP  CONT       主処理へ戻る
         END
```

次に、符号付きで乗算を計算するプログラムMULTSBSを示します(MULTSBS.casl)。メインルーチンの構造はMULTSAMPまったく同じであることに注意してください。このように、サブルーチンをうまく用いると、サブルーチンの内容を変えるだけで、プログラムをバージョンアップできるのです。

```
MULTSBS  START
         LD    GR3,A      Aの値をレジスタGR3に入れてMULTSUBに渡す
         LD    GR4,B      Bの値をレジスタGR4に入れてMULTSUBに渡す
         CALL  BMULTSB    サブルーチンMULTSUBの呼び出す
         LD    GR3,GR2    答(A×B)がレジスタGR2に戻るので、次の計算のために、GR3に格納
         LD    GR4,C      Cの値をレジスタGR4に格納
         CALL  BMULTSB    (A×B)×Cを計算
         ST    GR2,D      D ← (A×B)×C
         RET
A        DC    -25
B        DC    -4
C        DC    -10
D        DS    1
;
; サブルーチン(BMULTSB) GR2 ← GR3 × GR4 を符号付きで計算する
;
BMULTSB  XOR   GR0,GR0        後処理フラグ
         LAD   GR7,#FFFF      マスク
         LAD   GR6,1          1,定数
; GR3の符号判定
         LD    GR3,GR3        GR3のデータでフラグレジスタをセット
         JPL   BPROC          正なら次へスキップ
         JZE   RETSUB         ゼロなら計算しなくてもゼロ
         XOR   GR3,GR7        A(GR3)の符号
         ADDA  GR3,GR6        を反転
         XOR   GR0,GR6        GR0レジスタを反転
; GR4の符号判定
BPROC    LD    GR4,GR4        GR4のデータでフラグレジスタをセット
         JPL   CAL            正なら何もせずに計算へ
         JZE   RETSUB         ゼロなら計算しなくてもゼロ
         XOR   GR3,GR7      } B(GR4)の符号を反転
         ADDA  GR3,GR6      }
         XOR   GR0,GR6        GR0レジスタを反転
; 本計算
CAL      XOR   GR1,GR1        GR1←0, オーバーフローフラグ(0:していない,1:している)
         XOR   GR2,GR2        GR2←0, 解を格納するレジスタ
         LAD   GR5,16         GR5←16, 16回ループする(GR5はカウンタ)
LOOP     SRA   GR4,1          GR4(B)のビットを押し出す
         JOV   ADDVALUE       押し出された最下位ビットが1なら加算処理へ
CONT     SLA   GR3,1          Aを1ビット左へシフト
         SUBA  GR5,GR6        カウントダウン
```

```
          JNZ    LOOP          ゼロでなければLOOPへ
; GR0の値が1なら符号反転
FINISH    CPA    GR0,GR6
          JZE    MINUSC
RETSUB    RET                  ;プログラムの終了(答えはGR2に)
ADDVALUE  ADDA   GR2,GR3       解(GR2)に重み付けされたA(GR3)を加える
          JOV    SETFLAG       もし、オーバーフローしたならフラグをセット
          JUMP   CONT          主処理へ戻る
SETFLAG   LAD    GR1,1         オーバーフローフラグをセット
          JUMP   CONT          主処理へ戻る
MINUSC    XOR    GR2,GR7
          ADDA   GR2,GR6
          JUMP   RETSUB
          END
```

## 演習問題12-3

次のプログラムは、C←A÷Bをビット単位で効率よく行うものです。このプログラムの原理は、次の通りです。AをBで割った答えをC、余りをDとすると次の式が成立します。たとえば、294を11で割ると答えは26、あまりは8になります。

$A - C \times B = D \qquad 294 - 26 \times 11 = 8$

ここで、26を$2^n$の数の和に分解してみましょう。すると、$26 = 16 + 8 + 2 = 2^4 + 2^3 + 2^1$であることがわかります。つまり、式の形で書くと次のようになります。

$294 - (16 + 8 + 2) \times 11 = 8$

(16+8+2)の部分は、2進数の重み付けと対応させると、次のように書けます。

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | .... | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | | |
| | .... | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | | |
| 294 | − ( | | | | 16 + | 8 | + | 2 | | ) | × 11 =8 |
| 294 | − | 0 | 0 | 0 | 1 | 1 | 0 | 1 | $0_{(2)}$ | | × 11 =8 |

つまり、割り算294÷11の答えを求める作業は、何らかの処理を294と11に施して$11010_{(2)}$=26という数を求めることに他なりません。

そのためには、次のような処理を行えばよいでしょう。前処理として、次の処理をしておきます。

(1) A<Bなら、答=0、余=B
(2) A=Bなら、答=1、余=0
(3) A>Bなら、割り算の処理

A>Bという条件で、割り算の処理を進めます。最初に初期値として答を0にしておきます。まず、解の中でビットが1になるもっとも最上位の桁を求めます(上記の例では4ビット目。最初を0ビット目と数える)。そのためには、以下の処理をループで行えばよいでしょう。

294 − 0000000000000001(2) (=1) × 11 = 283 …・ A>Bなのでこれは必ず成立する
294 − 0000000000000010(2) (=2) × 11 = 272
294 − 0000000000000100(2) (=4) × 11 = 250
294 − 0000000000001000(2) (=8) × 11 = 206
294 − 0000000000010000(2) (=16) × 11 = 118
294 − 0000000000100000(2) (=32) × 11 = −58 …・ 負になるまで繰り返す

ここで294には32は11個含まれないが16は11(16×11)含まれることがわかります。そこで、32から16に戻します。

294 − 0000000000010000(2) (=16) × 11 = 118

この0000000000010000(2)が解の一部になるので、答に加えます。現時点における答は、「16=0000000000010000(2)」となります。

さて、次は、この残り(118)の中に11の要素がいくつあるかを調べます。そこで、さらに16から8に戻して次の計算を行います。

118 − 0000000000001000(2) (=8) × 11 = 30

118の中に11が8個あることが分かります。この値「8=0000000000001000(2)」を答に加えます。答は、「16+8=24」になります。さらに、8から4に戻して次の計算を行います。

30 − 0000000000000100(2) (=4) × 11 = −14

値が負なので、30の中に11は4つありません。さらに4から2に戻します。

30 − 0000000000000010(2) (=2) × 11 = 8

値が正なので30の中に11が2つあることがわかります。この値「2=0000000000000010(2)」を答に加えます。答は、「16+8+2=26」になります。

さらに2から1に戻して次の計算を行います。

8 − 0000000000000001(2) (=1) × 11 = −3

結局、もう8の中には11がありません。8が余りということになります。

この方針で作成したプログラムを以下に示します。空欄に適当な字句を入れてください。

```
;
; 効率の良い割り算 (DIVSUB.casl)
;
DIVMAIN   START
          LD    GR1,A
          LD    GR2,B
          CALL  DIV
          ST    GR3,C
          ST    GR1,D
          RET
A         DC    2940
B         DC    65
C         DS    1
D         DS    1
;
; 割り算を行うサブルーチン
;
DIV       CPA   GR1,GR2
          JMI   GR2LARGE      A<Bの処理
          JZE   SAME          A=Bの処理

; 割り算処理
          XOR   GR3,GR3       答←0
          XOR   GR6,GR6       0:定数
          LAD   GR5,1         GR5←0000000000000001
SHIFTLP   SLA   GR5,1
          SLA   GR2,1         GR2←GR2×2
          CPA   GR1,GR2
         [      1      ]
          SRA   GR5,1          1つ余分にシフトしたので戻す
          SRA   GR2,1
ADDLOOP   ADDA  GR3,GR5
          SUBA  GR1,GR2
NEXTBIT   SRA   GR5,1
          JZE   FINISH
          SRA   GR2,1
          CPA   GR1,GR2
         [      2      ]
          JUMP  ADDLOOP
FINISH    RET
GR2LARGE  XOR   GR3,GR3
          RET
SAME      LAD   GR3,1
          XOR   GR1,GR1
          RET
          END
```

## 12.4　再帰呼び出し

サブルーチンの中でさらにまた、自分自身のサブルーチンを呼ぶことができます。これを**再帰呼び出し**と呼びます。たとえば、再帰呼び出しで1～Nの和を求めるプログラムを作成することを考えてみましょう。そのためには、次の規則を使います。1からNまでの和をSUM(N)と定義すると、これは次のように定義できます。

SUM(N) = SUM(N－1) ＋ N　　(N≠0のとき)
SUM(N) = 0　　(N＝0のとき)

上の式(N≠0のとき)は、「1からNまでの和(1＋2＋3＋...＋(N－1)＋N)は、1からN－1までの和(1＋2＋3＋...＋(N－1))にNを加えたものである」ことを示しています。つまり、次の式が成立します。

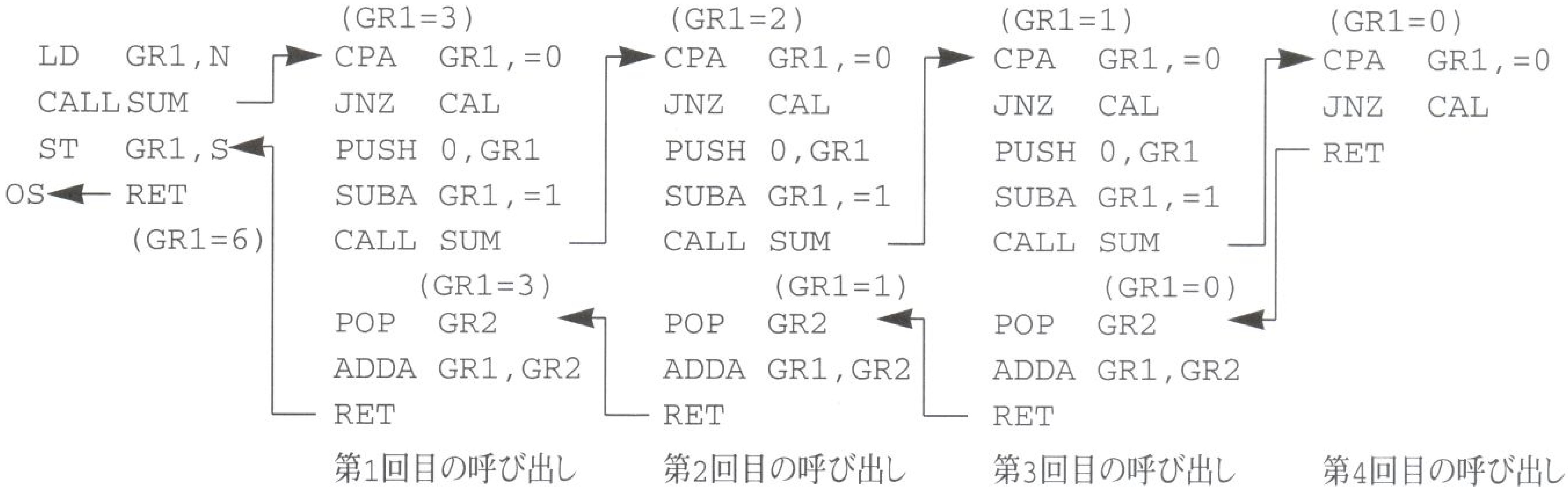
SUM(N)= SUM(N－1)　　　＋N
　　　= ( 1＋2＋3＋...＋(N－1) )　＋N

下の式(N＝0のとき)は、1から0までの和は0であるということを示しています。

この方法で作成したプログラムを以下に示します。SUMは、レジスタGR1にNの値を受け取り、1～Nの和を計算して、GR1にセットして返すサブルーチンです。

```
;メインルーチン
RSUM    START
        LD      GR1,N
        CALL    SUM
        ST      GR1,S
        RET
N       DC      100
S       DS      1
;
;1からnまでの和を求めるサブルーチン
;入力 GR1:n, 出力 GR1:和
SUM     CPA     GR1,=0       もし、N(GR1)がゼロかどうか
        JNZ     CAL          ゼロでなければCALへ
        RET                  ;ゼロならば、返却値もゼロ(SUM(0)=0)で終了
CAL     PUSH    0,GR1        Nの 値をプッシュして保存する
        SUBA    GR1,=1       GR1 ← N － 1 ( GR1－1 )
        CALL    SUM          SUMを呼ぶ。すなわち、 GR1 ← S( N－1 )を計算する。
        POP     GR2          Nの 値を復帰する
        ADDA    GR1,GR2      GR1 ← GR1 ＋ GR2 = S( N－1 )＋N
        RET                  ; GR1( S(N－1)×N )を返却値としてサブルーチンを終了
        END
```

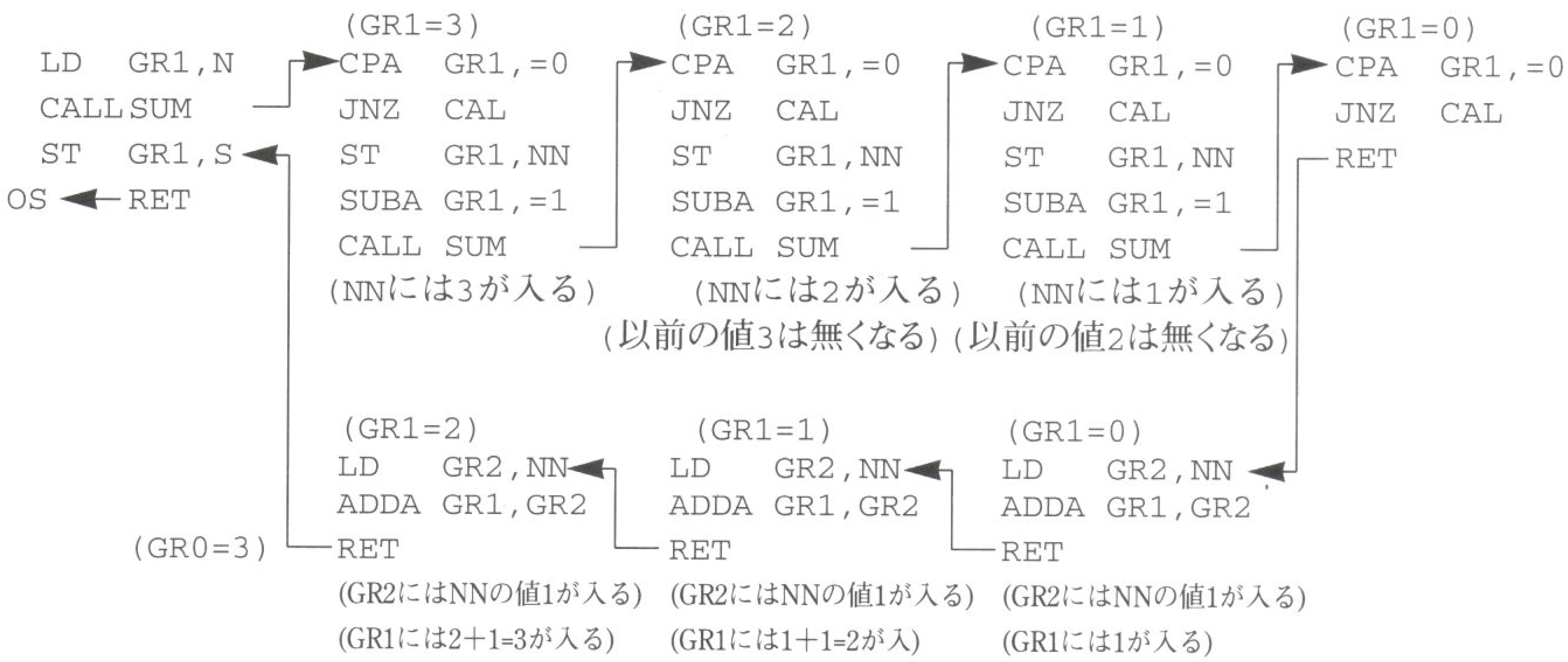
N＝3のときの実行の様子を次に示します。最初にSUMが呼び出されたときは、GR1＝3(N＝3)で呼び出されます。そして、GR1をPUSHしてスタックに格納し、GR1の値を1引いて(GR1＝2)にしてから、再びSUMを呼びます。GR1＝2ですからCALL SUMの実行が終われば、GR1には1＋2＝3が返ってきているはずです。そこで、スタックをPOPして先ほど退避しておいた値をGR2に受け取ります．この値は3のはずです。これをGR1(＝3)に加えれば6となり1＋2＋3が求められたことになります。



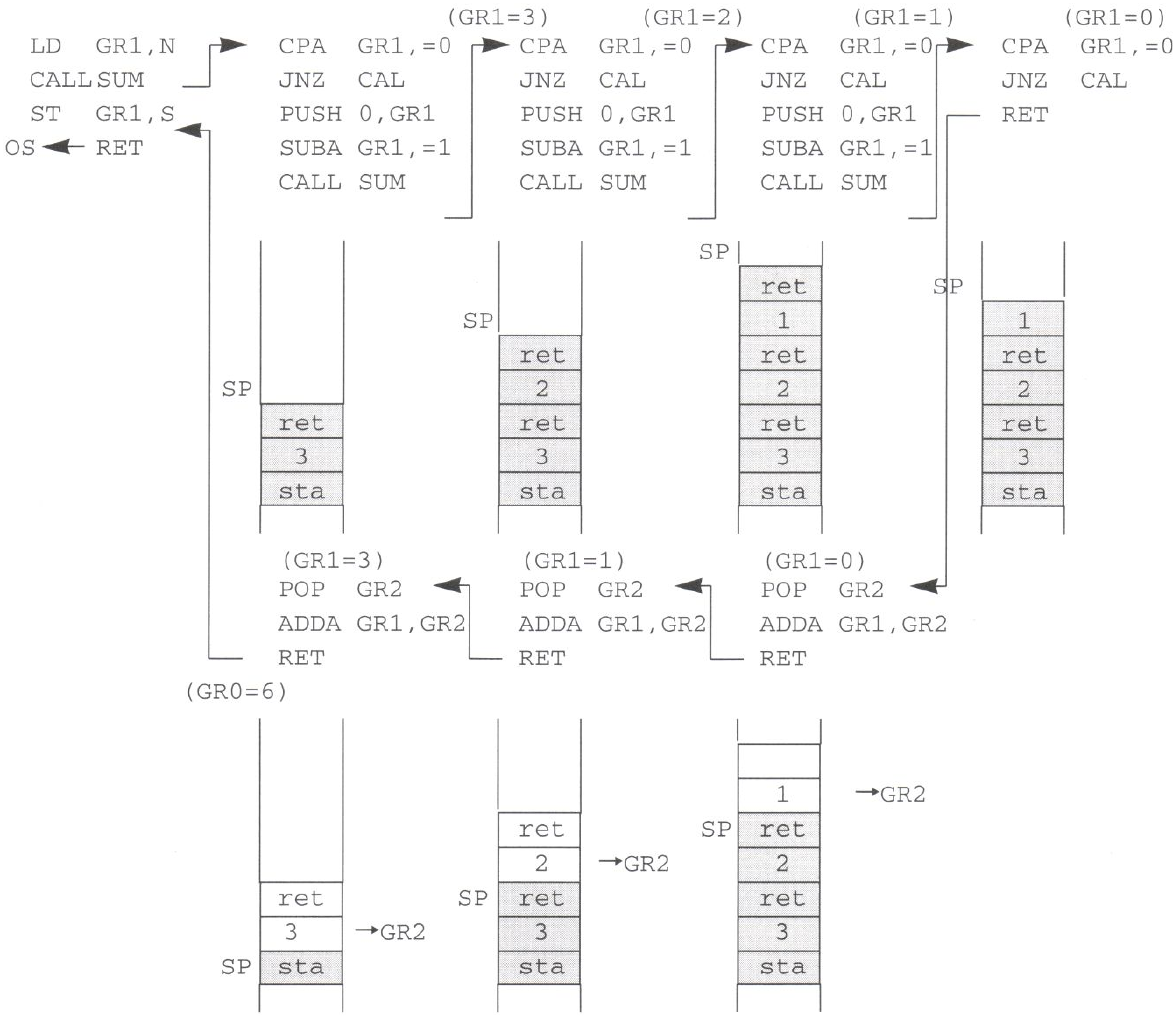

ところで、再帰呼び出しのプログラミングで難しいことは、自分自身のプログラムが実行途中に再び最初から実行されるということです。そのために、そのサブルーチンの中で、値を保持するために、ラベルでの領域を使うとうまくいかないことがあるのです。たとえば、上記のプログラムは、Nの値を保持するのに、スタックへ退避させていますが、これをもし、次のようにラベルNNに退避したらどうなるでしょうか。

```
SUM     CPA     GR1,=0      もし、N(GR1)がゼロかどうか
        JNZ     CAL         ゼロでなければCALへ
        RET                 ;ゼロならば、返却値もゼロ(Sum(0)=0)で終了
CAL     ST      GR1,NN      N の 値をNNに退避して保存する
        SUBA    GR1,=1      GR1 ← N － 1 ( GR0－1 )
        CALL    SUM         SUMを呼ぶ。すなわち、 GR0 ← S( N－1 )を計算する。
        LD      GR2,NN      N の 値を復帰する
        ADDA    GR1,GR2     GR1 ← GR1 ＋ GR2 = S( N－1 )＋N
        RET                 ;  GR0( S(N－1)＋N )を返却値としてサブルーチンを終了
NN      DS      1
        END
```

しかし、この方法では、NNの値は各サブルーチンの実行の中で次々に塗り替えられていって、最後にセットされた1がそのまま残り、正しく計算されていないことが分かります。



最初のプログラムは、この値をスタック上に保存していたので、問題なく計算が実行できたのです。この様子を示しましょう。メインプログラムの「ST GR1,S」のアドレスをsta、サブルーチンSUMの「POP GR2」のアドレスをretとします。スタックに積まれた値が再び取り出されていく様子が理解できると思います。




この様子は、付属のCASLシミュレータでも確認できます。ぜひ、実行させてみてください。

## ポイント

再帰手続きの中では、データはスタックに保存する。

## 演習問題12-4

再帰手続きを利用して、N!(階乗)を求めるサブルーチン(RFACT.casl)を作成してください。なお、N!は次のように定義できます。

```
N! = (N-1)! × N  (N≠0のとき)
N! = 1           (N=0のとき)
```

## 総合問題12

**[演習12-1]** 記憶域に入力されたテキストの中にある単語の個数をカウントするサブルーチンTEXTCNTを考えます。テキストの中には、英字と空白文字しかないものとします。たとえば、次のようになっているときの単語数は4です。

```
This is a pen
```

このプログラムは、テキストの先頭の文字から一文字ずつ取り出して、「空白(前の文字)+空白以外の文字(今の文字)」のパターンを見つけて、そのときだけ、単語数をカウントアップするという方法で、単語数をカウントしています。テキストの先頭の文字の前は、空白と仮定します。以下のプログラムの空欄に適当な単語を入れてください。

```
;
; テキスト中の単語数をカウントするプログラム(TEXTMAIN.casl)
;
TEXTMAIN    START
            CALL TEXTIN            テキスト入力
            CALL TEXTWC            単語数を取得する
            ST   GR0,COUNT         GR0→単語数
            RET                    ;OSへ戻る
COUNT       DS   1                 単語数
TEXT        DS   256               文字列入力エリア
LEN         DS   1
;
; テキスト入力サブルーチン
;
TEXTIN      IN   TEXT,LEN          TEXTへ文字列を読み込む
            [ 1 ] GR1,TEXT         TEXTのアドレスをGR1へ
            LD   GR2,LEN           TEXTの長さをGR2へ
            RET
;
; 単語数カウントサブルーチン
;
TEXTWC      LAD  GR3,1             1の定数として用いる
```

```
         XOR    GR0,GR0      個数をゼロにする
         XOR    GR4,GR4      フラグ←0、最初は単語の外にいる：0=単語の外、1=単語の中
LOOP     SUBA   GR2,GR3      個数をカウントダウン
         JMI    FINISH       個数が負になったらおしまい
        [      2      ]      アドレスGR1の内容をGR6へ(テキストから文字列を取り出す)
         CPA    GR6,=' '     その文字が空白か？
         JZE    SPACE        空白ならSPACEへ
        [ 3 ]   GR4,GR4      空白でないとき、単語の状態を調べる
         JPL    SKIPCNT      GR4>0、つまり、単語の中ならSKIPCNTへ
         ADDA   GR0,GR3      空白＋文字のときだから単語の区切りなので単語数(GR0)を＋1する
         LD     GR4,GR3      フラグ←1、つまり単語中にする
SKIPCNT  ADDA   GR1,GR3      アドレスGR1をカウントアップ
         JUMP   LOOP         LOOPへジャンプ
SPACE    XOR    GR4,GR4      空白なので、フラグを単語の外(=0)にする
        [      4      ]      アドレスGR1をカウントアップしてからLOOPへジャンプ
         JUMP   LOOP
FINISH   RET
         END
```

**[演習12-2]** 次に示すプログラムは金額の編集を行うサブルーチンMEDITSUBです。ある領域に格納された金額の文字データ(たとえば、'3000')を金額表示形式('¥3,000')に変更するものです。プログラム中のコメントを参考にして、空欄に適当な命令を入れてください。ただし、空欄の部分のコメントははずしてあるものもあります。

```
;
;  3000 ---> ¥3,000 に編集するプログラム (MEDIT.casl)
;
MEDIT   START
        IN    INDATA,INLEN    金額データの入力
        LAD   GR1,INDATA      GR1←INDATAのアドレス
        LD    GR2,INLEN       GR2←INDATAの長さ
        LAD   GR3,MONDATA     GR3←MODATAのアドレス
        CALL  MEDITSUB        金額編集
        ST    GR4,MONLEN      GR4→MONLEN(MONDATAの長さ)
        OUT   MONDATA,MONLEN
        RET
;
; 金額編集  (入力：GR1:元のデータのアドレス、GR2:元のデータの長さ、GR3:編集結果の領域 )
;           (出力：GR4:編集後の長さ)
;
MEDITSUB LAD  GR7,1           1:定数
         LD   GR4,GR7         GR4←1(MONDATAの長さ)
        [      1      ]       GR5←'¥'
         ST   GR5,0,GR3       '¥'→MONDATAの先頭へ
         ADDA GR3,GR7         GR3カウントアップ
; 「GR2(INDATAの長さ)-1」を3で割ったときの余りを求める
         LD   GR5,GR2         GR5←INDATAの長さ
         SUBA GR5,GR7         GR5カウントダウン
```

```
          LAD   GR6,3          GR6←3(定数)
LOOP1     CPA   GR6,GR5        GR5が3より小さい？
          JPL   EDITBODY       GR5=0,1,2ならばFOUNDへ
         [      2      ]       GR5←GR5-3
          JUMP  LOOP1
EDITBODY  LD    GR6,0,GR1      GR6←INDATAの次の文字
          ST    GR6,0,GR3      GR6→MONDATAの次の文字
         [      3      ]       MONDATAの文字数をカウントアップ
          SUBA  GR2,GR7        INDATAの長さをカウントダウン
          JZE   FINISH         INDATAの長さがゼロなら終了
          ADDA  GR1,GR7        INDATAの次の文字へ
          ADDA  GR3,GR7        MONDATAの次の文字へ
          SUBA  GR5,GR7        GR5カウントダウン
          JMI   COMMA          もし、GR5が負ならば「,」を挿入
          JUMP  EDITBODY       そうでなければ次の文字へ
COMMA     LAD   GR5,2          GR5←2 (3つのタイミングを取るため)
         [      4      ]       GR6←','
          ST    GR6,0,GR3      ','→MONDATAへ
          ADDA  GR4,GR7        MONDATAの文字数をカウントアップ
          ADDA  GR3,GR7        MONDATAの次の文字へ
          JUMP  EDITBODY
FINISH    RET
INDATA    DS    256
INLEN     DS    1
MONDATA   DS    256
MONLEN    DS    1
          END
```

# 第13章 実践問題編

前章まででCASL II、COMET IIに関する基本的かつ応用的な知識についてはほとんど網羅して学習しました。本章では、情報処理技術者試験などで出題される問題を解くことを中心に、試験テクニック的な内容を学習します。

情報処理技術者試験のCASL IIの問題では、非常に多くの分野から出題されています。基本情報技術者試験(旧名、第二種情報処理技術者試験)で出題された問題から、主なものを見てみると、たとえば、倍精度整数の加算(平成8年春、平成3年春)、キューの操作(平成10年春)、BCDコードの和(平成11年秋)、ビンゴの当たり検出(平成8年秋)、画像データの圧縮(平成5年春)などさまざまな種類の問題があります。

しかし、多くの場合、問題を解くのに十分な解説が示されていることが多く、たとえば、「画像圧縮」などの問題でも、基本的なCASL, COMETの知識とアセンブラプログラミングのテクニックを理解していれば、対応可能な問題がほとんどです。

まず、本章では、13.1節で、前章までで学習していないアセンブラテクニックについて説明し、13.2節で、実践的な問題の解き方について学習します。

## 13.1 実践的アセンブラテクニック

前述のように実際の試験問題では、出題される内容に関してあまり前提知識を要しませんが、それでも限られた試験時間の中では、ある程度の前提知識を有していることに越したことはありません。そこで、本節では、出題される可能性が高い分野について解説することにします。

### 13.1.1 多次元配列の処理

11章で、テーブル処理(表処理)を学習しました。そこでは、連続するいくつかの記憶域を指標レジスタを使って、ランダムにアクセスする方法を学習しました。この方法によって、「何番目」というデータを操作することができたのです。

しかし、世の中には、もっと複雑なデータが存在します。たとえば、ある中学校のクラスの人数を管理することを考えてみましょう。各学年のクラスが1組から5組まであるとすると、その人数のデータは次の表として示されます。

表 13-1 ある中学校の各クラスの人数

| 学年＼組 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 40 | 43 | 44 | 41 | 49 |
| 2 | 45 | 39 | 43 | 40 | 46 |
| 3 | 44 | 40 | 43 | 47 | 41 |

表13-1を見ると、3行5列にデータを並べていることが分かります。そして、2年3組に43人いることが分かりますが、この例のように、x 年 y 組というように、表の中のデータを選択するのに、2つの値(x,y)が必要になります。

このような表を、一般的なプログラミング言語の言葉では、**2次元配列**と呼びます。さらに、内部のデータを特定するのに(x,y,z)のデータが必要になる**3次元配列**やそれ以上の**多次元配列**(4次元配列、5次元配列、...)が存在します。

一方、いままでのテーブル(表)を、この言葉に合わせて、**1次元配列**と呼ぶこともあります。また、表の中の位置を特定する値(x, y, zなど)のことを**添字**と呼びます。

CASL IIでは、多次元配列は、どのように処理されるのでしょうか。実は、多次元配列という概念はCOMET IIの機械語にはありません。COMET IIでは、指標レジスタを用いるテーブル(1次元配列)しか処理できません。COMET IIで、多次元配列を扱う場合には、テーブルを使って、位置の計算を行うようにします。たとえば、表13-1のデータを格納するのには、次のようにすべてを連続したテーブルに格納します。

```
NUMBER    DC     40,43,44,41,49,45,39,43,40,46,44,40,43,47,41
```

x 年 y 組の人数は、先頭から 5×(x−1)＋(y−1) 番目(最初を0番目とする)のデータとなります。たとえば、2年3組なら、5×(2−1)＋(3−1)＝5×1＋2＝7番目のデータとなります。

このデータを取り出すプログラムの一般形は次のようになるでしょう。実際には、プログラム中のxは1〜3のどれかの値に、yは1〜5のどれかの値に置き換えてください。

```
LAD   GR0,1              GR0←1(計算用定数)
LAD   GR1,x              GR1←x
SUBA  GR1,GR0            GR1←(x−1)
LAD   GR2,y              GR2←y
SUBA  GR2,GR0            GR2←(y−1)
LD    GR3,GR1            GR3←(x−1)
SLA   GR3,2              GR3←4×(x−1)
ADDA  GR1,GR3            GR1←5×(x−1)
ADDA  GR1,GR2            GR1←5×(x−1)＋(y−1)
LD    GR4,NUMBER,GR1     GR4←(x,y)の値
```

ところで、上記のプログラムは、2次元配列の各行の先頭アドレスを計算するために、5×(x−1)という計算を行っています。ループの中などでこのような計算を毎回行っていたのでは、効率が悪くなります。そこで、次のように各行の先頭アドレスを保持することも考えられます(ARRAYA.casl)。これは、2年3組の人数をGR4にセットするプログラムです。

```
ARRAYA    START
          LAD  GR0,1              GR0←1(計算用定数)
          LAD  GR1,2              GR1←x
          SUBA GR1,GR0            GR1←(x−1)
          LD   GR3,NUMBERA,GR1    GR3←x行目の先頭アドレス
          LAD  GR2,3              GR2←y
          SUBA GR2,GR0            GR2←(y−1)
          ADDA GR3,GR2            GR3←x行目の(y−1)番目のアドレス
          LD   GR4,0,GR3          GR4←(x,y)の値
          RET
NUMBER0   DC    40,43,44,41,49
NUMBER1   DC    45,39,43,40,46
NUMBER2   DC    44,40,43,47,41
NUMBERA   DC    NUMBER0,NUMBER1,NUMBER2
          END
```

NUMBERAの表の各要素は、それぞれテーブルの0行目、1行目、2行目の先頭要素のアドレスになっています。値 x から、まず、x行めの先頭要素のアドレスを取得し、その後、その値にy－1を加えて目的とする位置のアドレスを算出します。

また、2次元配列の例ではないのですが、過去の例の中でいくつか出題されている問題が、16語の連続した語の中の各語のビットを用いた問題です。たとえば、次の例は、連続した8語を使って「7」を表現した例です。

```
0011111111111100
0011000000001100
0000000000001100
0000000000011000
0000000000110000
0000000001100000
0000000011000000
0000000011000000
```

このデータは、次のようなDC命令で定義できます。

```
DC   #3FFC,#300C,#000C,#0018,#0030,#0060,#00C0,#00C0
```

このような問題は、多次元配列の処理ではなく、テーブル操作とビット操作の組み合わせ問題であるといえます。

### 13.1.2 データ構造の処理

リスト処理や二分木の問題も慣れていないと戸惑うことが多い問題です。例として携帯電話の電話帳を扱うプログラムについて考えてみましょう。携帯電話の電話帳には，読みや漢字氏名などをランダムに登録し、またランダムに削除することができます。

しかし、登録はランダムでも、呼び出すときには、ちゃんと「あいうえお順」になって表示されます。リスト処理を使うとこのような処理が簡単に実現できます。**リスト処理**とは、簡単に説明すると、データを順番に配置するのではなく、データの中に次のデータの位置の情報を含めることによって、ランダムにデータを格納しても正しい順番で処理を行うテクニックの総称です。

簡単な例で、リスト処理について説明しましょう。いま、データが次のようにランダムに配置されているとします。

```
'ｳﾁﾀﾞ ｻﾄｼ','ﾖｺﾊﾏ ﾊﾅｺ','ｶﾅｻﾞﾜ ﾀﾛｳ','ﾊｺﾈ ｼﾞﾛｳ','ｲｽﾞ ﾕｳﾀﾛｳ'
```

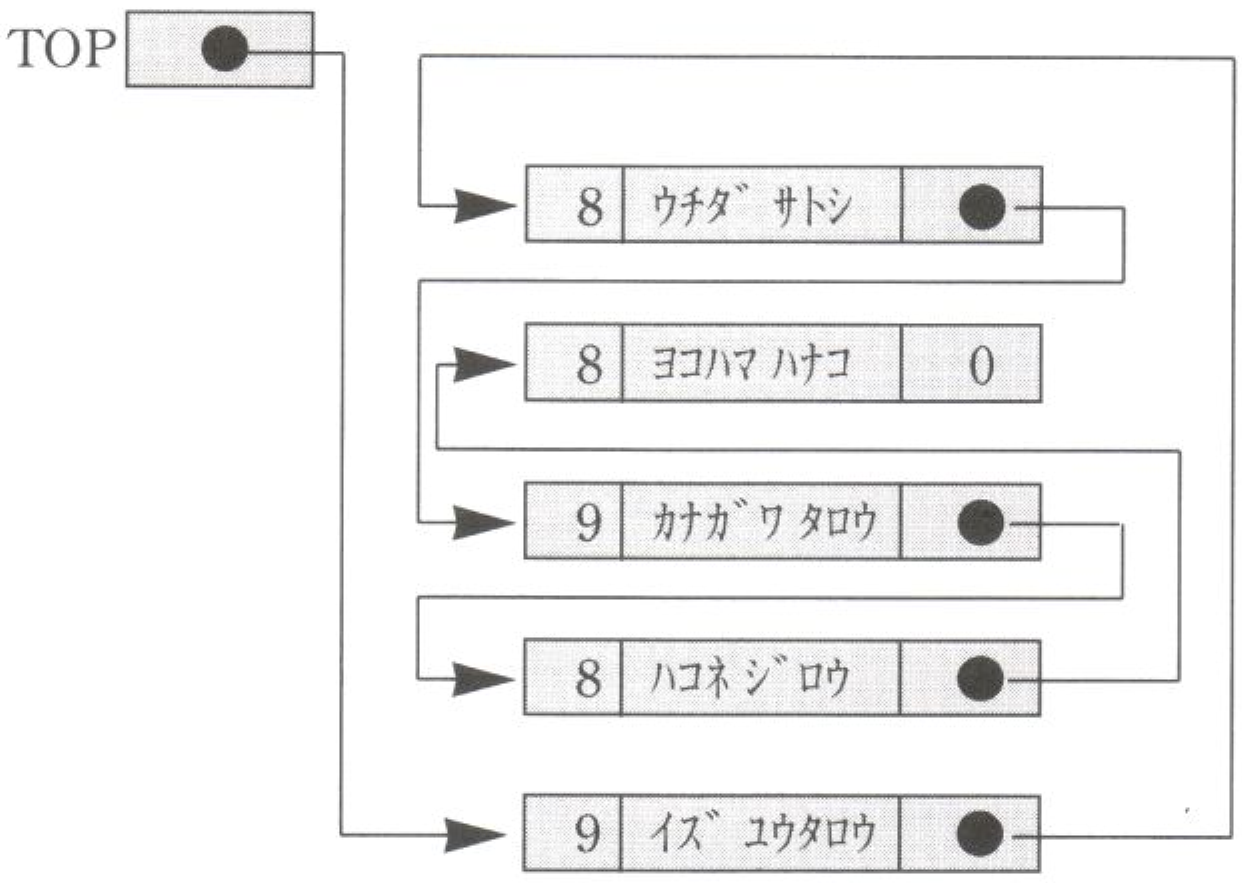
この各データを次のように「あいうえお順」に表示することを考えます。

```
ｲｽﾞ ﾕｳﾀﾛｳ
ｳﾁﾀﾞ ｻﾄｼ
ｶﾅｶﾞﾜ ﾀﾛｳ
ﾊｺﾈ ｼﾞﾛｳ
ﾖｺﾊﾏ ﾊﾅｺ
```

そのために、各データに名前の文字数と次のデータが格納されているのアドレスを付加します。具体的には、次のようにデータを定義します。

```
TOP      DC   TEL05                      先頭のデータのアドレス
TEL01    DC   8,'ｳﾁﾀﾞ ｻﾄｼ',TEL03
TEL02    DC   8,'ﾖｺﾊﾏ ﾊﾅｺ',0             ゼロはデータの最後を意味する
TEL03    DC   9,'ｶﾅｶﾞﾜ ﾀﾛｳ',TEL04
TEL04    DC   8,'ﾊｺﾈｼﾞﾛｳ',TEL02
TEL05    DC   9,'ｲｽﾞ ﾕｳﾀﾛｳ',TEL01
```

このデータを図示すると、次のようになります。また、このようにデータに構造を持たせたものを**データ構造**と呼びます。




つまり、TOPは最初のデータの位置を示しており、TOPが指しているアドレスには、「ｲｽﾞ ﾕｳﾀﾛｳ」の文字数である9が格納されており、その次の位置から文字列「ｲｽﾞ ﾕｳﾀﾛｳ」が配置されています。なお、分かりやすいように半角かな文字を用いていますが、漢字は使うことができないので注意してください(＊1)。また、各文字は1バイトに収められ、上位8ビットの全てのビットが0になることに注意してください。したがって、「ｲ」の位置から9(文字数分)だけ加算した位置に、次のデータのアドレスが入っています。「ｲｽﾞ ﾕｳﾀﾛｳ」の例では、この部分にTEL01が入っており、これが次のデータ「ｳﾁﾀﾞ ｻﾄｼ」の先頭アドレスを指しています。このように各データの先頭アドレスを次々にたどっていきますので、データがバラバラに配置されていても色々な処理ができるのです。

参考のために、各データをたどって、それを順番に表示するプログラムを以下に示します。

```
TELNO    START
         LAD  GR1,1           GR1←1(定数)
         LD   GR2,TOP         GR2←TOP(先頭アドレス)
LOOP     JZE  FINISH          それがゼロなら終了
         LD   GR3,0,GR2       GR3←文字数
         ADDA GR2,GR1         GR2←名前の先頭アドレス
         CALL PRINT           名前を表示する
         ADDA GR2,GR3         GR2←次のアドレス
         LD   GR2,0,GR2       リンクをたどる
         JUMP LOOP
FINISH   RET                  ;OSへ戻る
PRINT    PUSH 0,GR3           レジスタの退避
         PUSH 0,GR2
```

(＊1) 本書で提供する処理系は半角かな文字を使えるようにしてあります。もちろん、コメントはプログラムの一部ではありませんので漢字も半角かな文字も使えます。

```
          ST    GR3,PRLEN     PRLENへ文字数をセット
          XOR   GR4,GR4       GR4クリア：PRのインデックス
COPYLOOP  LD    GR0,0,GR2     データを1文字取り出して
          ST    GR0,PR,GR4    PRに書き出す
          ADDA  GR4,GR1       GR4←GR4+1 PRのインデックス
          ADDA  GR2,GR1       GR2←GR2+1 名前のインデックス
          SUBA  GR3,GR1       GR3←GR3−1 文字数カウント
          JPL   COPYLOOP      まだ、文字があればループする
          OUT   PR,PRLEN      名前の出力
          POP   GR2           レジスタ復元
          POP   GR3
          RET
TOP       DC    TEL05         先頭のアドレス
TEL01     DC    8,'ｳﾁﾀﾞ ｻﾄｼ',TEL03
TEL02     DC    8,'ﾖｺﾊﾏ ﾊﾅｺ',0
TEL03     DC    9,'ｶﾅｶﾞﾜ ﾀﾛｳ',TEL04
TEL04     DC    8,'ﾊｺﾈ ｼﾞﾛｳ',TEL02
TEL05     DC    9,'ｲｽﾞ ﾕｳﾀﾛｳ',TEL01
PR        DS    256
PRLEN     DS    1
          END
```

実行結果は次の通りです。

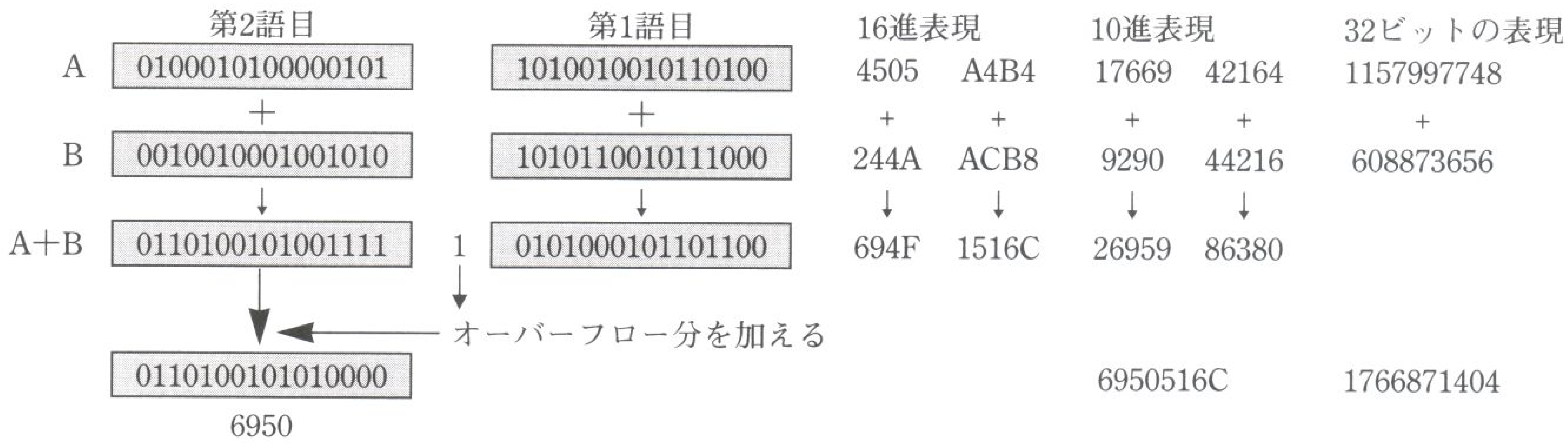
```
ｲｽﾞ ﾕｳﾀﾛｳ
ｳﾁﾀﾞ ｻﾄｼ
ｶﾅｶﾞﾜ ﾀﾛｳ
ﾊｺﾈ ｼﾞﾛｳ
ﾖｺﾊﾏ ﾊﾅｺ
```

### 13.1.3 多倍精度の処理

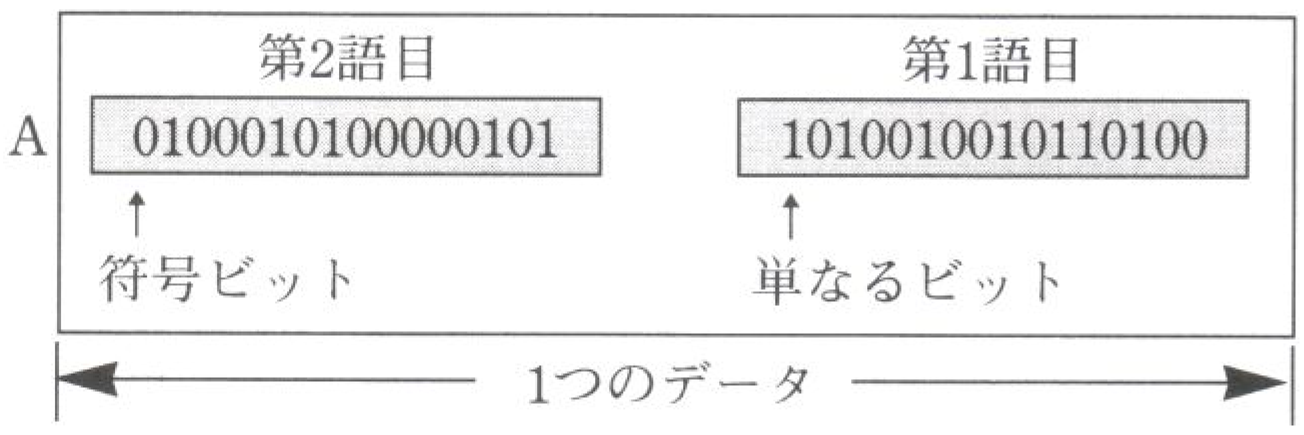
COMET IIは、1語が16ビットのコンピュータなので、扱える整数の大きさは−32768〜32767あるいは0〜65535という少ない数しか扱えません。これを倍の32ビット(2語)にすると−2147483648〜2147483647あるいは0〜4294967295にすることができます。COMET IIの16ビットの演算機能を利用して、32ビットの計算を行うことを**倍精度計算**と呼びます(もっとも、一般のCPUでは32ビットが標準ですから、この機能は一般のCPUから見た場合には普通の計算ということになりますが)。

それでは、倍精度の加算について考えてみましょう。倍精度データは、前述のように、2語占有しているわけですね。また、この2語は処理しやすいように連続して配置します。なお、ここでは、A≧0, B≧0であることを前提として、A+Bについて考えてみます。

ここでは、A=1157997748, B=608873656としましょう。Aの値を16進数で表現すると4505A4B4(16)ですから16進数の下位4桁(A4B4(16))を第1語目、上位4桁(4505(16))を第2語目に配置します。




まず、上記の図のように、AとBの第1語目を論理加算(ADDL)します。この計算を算術加算(ADDA)にしてしまいますと、符号ビット(最上位ビット)を処理してしまいますので、正しく計算できなくなります。符号ビットとして扱われるビットは、連続する2語の最上位ビットだけです。




第1語目の最上位ビットは単なる普通のビットとして扱う必要があります。第2語目の加算では、最上位ビットを符号ビットとして扱い、かつ、オーバーフローを検出したいので、算術加算します。もし、第1語目の計算でオーバーフローしたらその分(+1)を第2語目の答えに加えます。第2語目の加算でオーバーフローが起きたら、それは32ビット計算でのオーバーフローを意味します。

```
LONGINT  START
         XOR    GR0,GR0       第1語目の桁上がりを保持する領域
         LD     GR1,A1        GR1←Aの1語目
         LD     GR2,B1        GR2←Bの1語目
         ADDL   GR1,GR2       GR1←GR1+GR2(符号を無視)
         JOV    PLUSONE       オーバーフローがあればPLUSONEへ
NEXT     LD     GR3,A2        GR3←Aの2語目
         LD     GR4,B2        GR4←Bの2語目
         ADDA   GR3,GR4       GR3←GR3+GR4(符号付きで計算)
         JOV    OVERFLOW      オーバーフローしているか
         ADDA   GR3,GR0       下位桁の繰り上がりを加算
         JOV    OVERFLOW      オーバーフローしているか
         ST     GR1,C1        GR1→C1
         ST     GR3,C2        GR3→C2
         XOR    GR0,GR0
         ST     GR0,OVF       0 → OVF
         JUMP   EXIT
OVERFLOW LAD    GR0,1         オーバーフローしたので
         ST     GR0,OVF       1 → OVF
EXIT     RET
PLUSONE  LAD    GR0,1         桁上がりのための1をセット
         JUMP   NEXT
A2       DC     #4505
A1       DC     #A4B4
B2       DC     #244A
B1       DC     #ACB8
C2       DS     1
C1       DS     1
OVF      DS     1
         END
```

# 13.2 実践問題

## 13.2.1 実践的な問題の解き方

情報処理技術者試験の問題を限られた時間内で解くためには、自分でプログラムを作る能力よりも、他人が作成したプログラムを理解する能力が要求されます。つまり、問題に提示されているプログラムを、それを作った作者の考え方に沿って理解しなければなりません。プログラムの作り方には、非常に多くのバリエーションがあります。自分の考え方に固執していると、試験問題で出題されているプログラムの全体像が見えてこないことがあります。

試験を解くときには、時間との勝負になることが多いので、効率よく問題を解く必要があります。そのためには、たとえば、次のような手順を踏むのも一つの方法です。

(1) 問題をよく読んで、内容を理解する。そのときに、注意点や、よく分からないところはマークをしておく。

(2) プログラムを読むときは、ラベルの位置、汎用レジスタの役割、初期化の有無、に注意する。そして、プログラム中のループやジャンプの状況などを視覚的に明確にするために、プログラムリスト上に矢印などを書き込む。特に、COMET IIになって、汎用レジスタの個数が増えた。場合によっては、効率のために、単に定数を入れるだけの汎用レジスタもあるかもしれない。

(3) プログラムの動作が分かりにくいときは、適当なデータを想定してトレースを試みる。選択式の場合には、この方法で解を得ることもできることがある。

(4) データの格納状況や汎用レジスタの使われ方など、可能な限り絵を描く。特にアドレスを扱う場合には、対象としているデータがアドレスそのものなのか、どこかの基準アドレスからの変位なのか、問題にはっきりかかれていない場合もあるので注意する。

しかし、いずれにせよ、なるべく多くの問題を実際に解いてみることが、一番の学習方法でしょう。以下に、過去の問題(第二種情報処理技術者試験問題)の中から代表的な問題を選択し、具体的な解き方を示しますので、参考にしてください。

なお、過去の問題は、古いCASL, COMET用に作られたものなので、これをCASL II, COMET IIに書き直してあります。

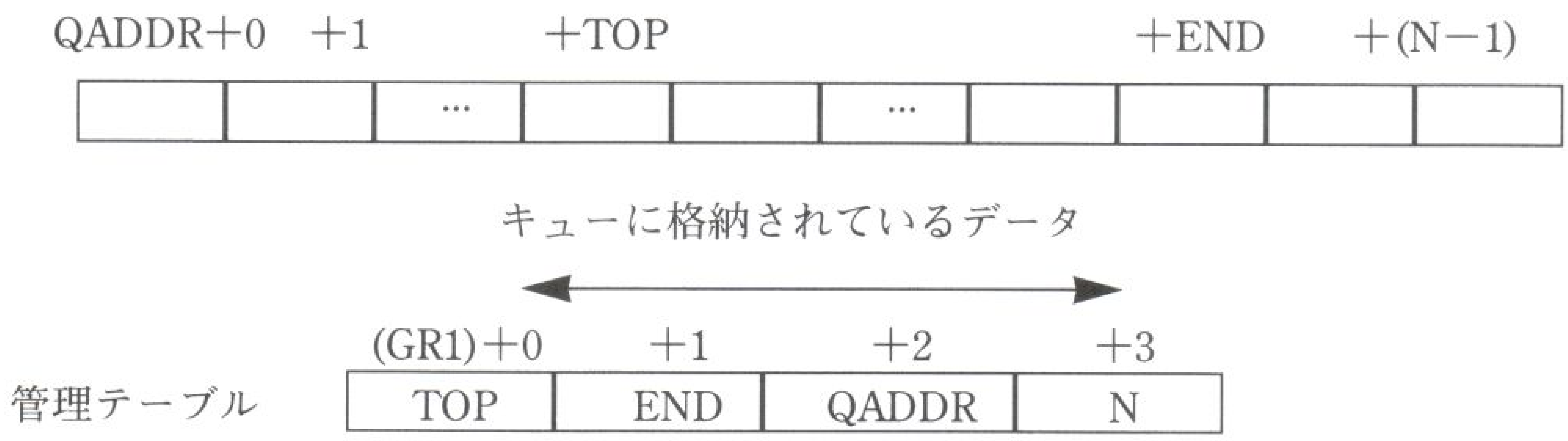
## 13.2.2 平成10年度春期問題(CASL II用に改変)

次のアセンブラプログラムの説明及びプログラムを読んで、設問に答えよ。

[プログラムの説明]

キューへデータを格納する副プログラムQINと、キューからデータを取り出す副プログラムQOUTである。キューのデータは、格納した順に取り出される。

(1) キューを、図に示すようにN個(1<N<32768)の要素からなる領域と管理テーブルで実現する。




(2) キューを実現する領域の末尾(QADDR+(N−1))は先頭(QADDR)につながっているものとみなす。すなわち、領域の末尾を越えてデータを格納しようとした場合、領域の先頭に戻って格納を試みる。

(3) キューへデータを格納したいプログラムは、GR1に管理テーブルの先頭番地を、GR0に格納するデータを設定して、QINを呼ぶ。QINは、ENDの位置にデータを格納し、ENDの値を更新した後、GR0に0を設定して処理を終える。ただし、既にN−1個のデータが格納済みならば、GR0に−1を設定して処理を終える。

(4) キューからデータを取り出したいプログラムは、GR1に管理テーブルの先頭番地を設定してQOUTを呼ぶ。QOUTは、TOPの位置のデータをGR0に設定し、TOPの値を更新した後、処理を終える。ただし、キューが空ならば、GR0に−1を設定して処理を終える。

(5) TOPとENDは0に初期化されているものとする。

(6) QINとQOUTは、呼ばれたプログラムに戻るとき、汎用レジスタGR2, GR3の内容を元に戻す。

[プログラム]

```
QIN      PUSH 0,GR2
         PUSH 0,GR3
        [      a      ]      END → GR2
         ADDA GR2,=1
         CALL ADJ
         CPL  GR2,0,GR1      GR2=TOP?
         JZE  QFULL          キューが一杯
         LD   GR3,2,GR1
        [      b      ]
         ST   GR0,0,GR3      データの格納
         ST   GR2,1,GR1      ENDの更新
         XOR  GR0,GR0
         JUMP IFIN
QFULL    LAD  GR0,−1
IFIN     POP  GR3
         POP  GR2
         RET
QOUT     PUSH 0,GR2
         PUSH 0,GR3
         LD   GR2,0,GR1      TOP→GR2
         CPL  GR2,1,GR1      TOP=END?
```

```
        JZE   QEMPTY        キューが空
        LD    GR3,2,GR1
        ADDL  GR3,0,GR1
        LD    GR0,0,GR3     データの設定
        ADDA  GR2,=1        TOP+1→GR2
        CALL  ADJ
      [        c       ]
        JUMP  OFIN
QEMPTY  LAD   GR0,-1
OFIN    POP   GR3
        POP   GR2
        RET
ADJ     CPL   GR2,3,GR1     領域の末尾処理
        JMI   AFIN
      [        d       ]
AFIN    RET
CONTBL  DC    0,0,QUEUEA,100
QUEUEA  DS    100
        END
```

設問 プログラム中の[ ]に入れる正しい答えを、解答群の中から選べ。

aに関する解答群

ア LD GR2,0,GR1　　イ LD GR2,1,GR1　　ウ LD GR2,2,GR1
エ LD GR2,3,GR1

bに関する解答群

ア ADDA GR2,1,GR1　　イ ADDA GR3,1,GR1　　ウ ADDA GR2,1,GR3
エ LAD GR3,1,GR3　　オ SUBA GR2,1,GR3　　カ SUBA GR3,1,GR1

cに関する解答群

ア ST GR2,0,GR1　　イ ST GR2,0,GR3　　ウ ST GR2,1,GR1
エ ST GR2,1,GR3

dに関する解答群

ア ADDA GR2,2,GR1　　イ LD GR2,3,GR1　　ウ XOR GR2,GR2
エ SUBA GR2,2,GR1

**解き方**

(1) 問題の理解

問題文をよく読んで理解する。

(2) 図の理解

図から、キューの先頭アドレスがQADDRであり、TOP〜END−1までにデータが格納されている

ことを読み取り、TOPやENDの値は管理テーブルの中に格納されていることを読みとる必要があります。また、TOP, ENDの値は先頭からの変位であり、アドレスではないことを読み取る必要があります。それは、キューの上の表示(QADDR＋0)の後のTOPとENDの前に＋が付いていることから類推できます。TOP, ENDにアドレスが格納されていると勘違いしてしまうと、解答群の中に正しい選択肢がなく、混乱するかも知れません。

(3) aの穴埋めを行う

別に最初の設問から穴を埋めなくても良いのですが、プログラムのaの部分にコメントがあるので、これを参考にすればすぐに解けそうです。それは、「END→GR2」というコメントです。このヒントから、ENDの値をGR2へ持ってくれば良いことがすぐに分かります。ENDは、管理テーブルの1番目です。管理テーブルのアドレスはGR1に格納されていますから、指標レジスタGR1を用いて、「1,GR1」とすれば、この実効アドレスはENDの格納されているアドレスになります。

したがって、「LD GR2,1,GR1」となることがすぐに分かります。答えはイとなります。

(4) その後のプログラムを分析する

aの穴埋めが分かったので、その次へ進みます。aの次の命令は、「ADDA GR2,＝1」です。これは、GR2(ENDの位置)を一つ進めています。おそらく、データを追加するための措置であることが考えられます。そして、次にサブルーチンADJを呼び出します。これは、そのコメントを見ると、「領域の末尾処理」とありますので、おそらくENDが末尾を越えた処理であろうと思われます。GR2の値が管理テーブルの3番(データ数が入っている)よりも小さければRETしています。穴埋めdは、ENDの位置を示すGR2がキューの領域を越えた時の処理であると考えられますので、GR2の値はそのときにはゼロに戻さなければなりません。これは、「プログラムの説明(2)」の「キューを実現する領域の末尾(QADDR＋(N－1))は先頭(QADDR)につながっているものとみなす。すなわち、領域の末尾を越えてデータを格納しようとした場合、領域の先頭に戻って格納を試みる」の部分で説明されていますが、キューのことを多少知らないと、直ちにこの処理を思い付くことは難しいかも知れません。GR2をキューの先頭、すなわちゼロにするには、「XOR GR2,GR2」を選びます。dの答えは、ウとなります。

(5) 上から8行目、「CPL GR2,0,GR1」以降の処理を考える

サブルーチンADJから戻ってきた後の処理を考えましょう。ADJから戻ってくると、すぐにキューがいっぱいになっているかどうかチェックしています。その後にある穴埋めbの次に「ST GR0,0,GR3」があります。これは、GR0の内容をキューにセットしていることがコメントからも読み取れます。すると、GR3には、ENDのアドレスが入っていなければなりません。ENDのアドレスは、QADDR＋ENDです。穴埋めbの上の命令を見てみると、「LD GR3,2,GR1」です。これは、管理テーブルの2つめ、すなわち、QADDRのアドレスをGR3に入れているのです。それならば、このGR3に、管理テーブルの1番目のデータ(END)を加えれば良いのですから、bは「ADDA GR3,1,GR1」となります。bの答えはイとなります。

(6) 最後のcを埋める

最後のcの選択欄を見ると、すべて「ST GR2,～」となっています。ということは、GR2の値を主記憶装置のどこかに書き出す処理であることが分かります。その上のプログラムを分析すると、次のよ

うになります。

```
LD   GR3,2,GR1     「2＋GR1(QADDR)」→GR3、つまりキューの先頭アドレスQADDRをGR3にセット
ADDL GR3,0,GR1     「0＋GR1(TOP)」をGR3に加える。つまり、TOPの位置をGR1にセット
LD   GR0,0,GR3     TOPの位置の内容をGR0に転送
ADDA GR2,=1        TOP＋1→GR2
CALL ADJ           領域の末尾処理
```

さて、キューの処理を考えると、キューから取り出す時はTOPの値です。取り出した場合、TOPの位置は次のTOP＋1になっていますが、それを管理テーブルに書き出していません。そこで、cは、「ST GR2,0,GR1」となります。cの答えはアとなります。

解答

aイ　　bイ　　cア　　dウ

### 13.2.3 平成11年度秋期問題(CASL II用に改変)

次のアセンブラプログラムの説明及びプログラムを読んで、設問1, 2に答えよ。

[プログラムの説明]

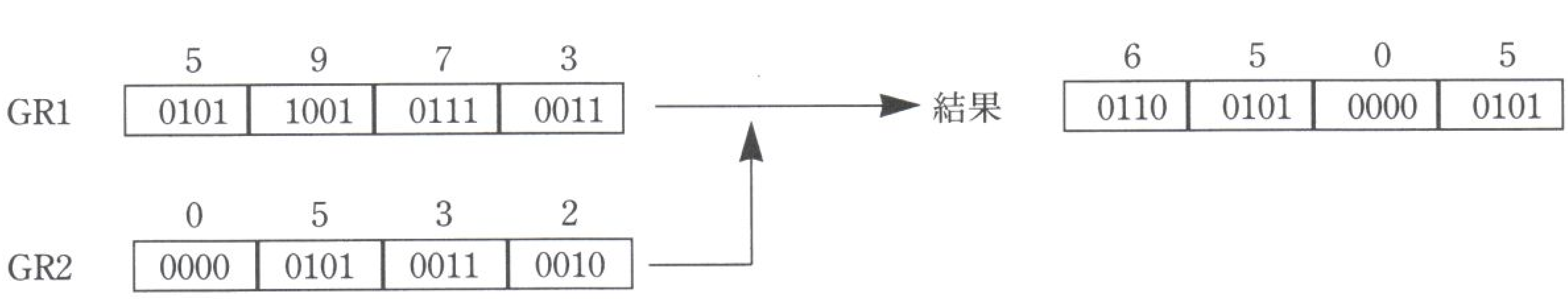
副プログラムADDPは、主プログラムから、BCDコードで表現された二つの整数a,b(a≧0,b≧0)を受け取り、aとbの和をBCDコードとして求める。

(1) BCDコードとは、10進数の値に対して、その10進数の各けたを4ビットの2進数で表現したものである。たとえば、10進数の5973をBCDコードで表現すると、次のとおりになる。

| 千の位 | 百の位 | 十の位 | 一の位 |
|---|---|---|---|
| 5 | 9 | 7 | 3 |
| ↓ | ↓ | ↓ | ↓ |
| 0101 | 1001 | 0111 | 0011 |

(2) aとbは共に4けた(16ビット)であり、それぞれGR1とGR2に格納されて、主プログラムから渡される。

(3) 副プログラムADDPの実行例を図に示す。




(4) 主プログラムに戻るとき、汎用レジスタGR2,GR3の内容を元に戻す。

[副プログラムADDP]

```
ADDP    START
        ST    GR1,A
        ST    GR2,B
        PUSH  0,GR3
        XOR   GR3,GR3      } 結果の領域の初期化
        ST    GR3,RES      }
LOOP1   AND   GR1,MASK1    整数aから4ビットを抽出
        AND   GR2,MASK1    整数bから4ビットを抽出
        XOR   GR0,GR0
        ADDA  GR1,GR2      4ビットの加算
        CPA   GR1,TEN      結果≧10
        JMI   NOTOVR       NO
        SUBA  GR1,TEN      結果から10減算
        LAD   GR0,1        けた上げの設定
NOTOVR  SLL   GR1,0,GR3    4ビットの加算結果を元のビット位置へ
        OR    GR1,RES      中間結果に併合
        LAD   GR3,4,GR3
        CPA   GR3,=16      全桁終了?
        JZE   FIN1         YES
        ST    GR1,RES      中間結果を格納
        LD    GR1,A        } 整数aの次の4ビットを右端へ
        SRL   GR1,0,GR3    }
        LD    GR2,B        } 整数bの次の4ビットを右端へ
        SRL   GR2,0,GR3    }
        ADDA  GR1,GR0      けた上げ分を加算
        JUMP  LOOP1
FIN1    POP   GR3
        LD    GR2,B
        RET
MASK1   DC    #000F
TEN     DC    10
A       DS    1
B       DS    1
RES     DS    1
        END
```

設問1 次の文中の[　]に入れる正しい答えを、解答群の中から選べ。

主プログラムから渡されたGR1とGR2の内容が次の通りであった場合、ADDPはその実行後、GR0に[ a ]をGR1に[ b ]を設定して、主プログラムに返す。

```
GR1 = 0101 0010 1001 1000
GR2 = 0100 1000 0101 0011
```

解答群

ア 0000 0000 0000 0000 イ 0000 0000 0000 0001 ウ 0000 0001 0101 0001

エ 0100 1000 0101 0011 オ 0101 0010 1001 1000 カ 1001 1001 1001 1001

キ 1010 0001 0101 0001 ク 1111 1111 1111 1111

設問2 副プログラムADDPを使用して、BCDコードで表現された二つの整数m,n(m≧0, n≧0)に対し、ある演算を行う副プログラムCALCを作成した。

(1) mとnは、それぞれGR1とGR2に格納されて、主プログラムから渡される。

(2) 演算結果をGR1に格納する。

(3) 主プログラムに戻るとき、汎用レジスタGR2, GR3の内容を元に戻す。

表は、主プログラムから渡されるmとnの値、主プログラムに返すGR1の値、及び副プログラムADDPを呼び出す回数を示す。ただし、表中および解答群中の値はすべて10進数とする。表中の［　］に入れる正しい答えを、解答群の中から選べ。

表 各値の対応

| 主プログラムから渡される値 | | 主プログラムに返すGR1の値 | ADDPを呼び出す回数 |
|---|---|---|---|
| m | n | | |
| 123 | 45 | c | d |
| 51 | 3579 | e | f |

[副プログラムCALC]

```
CALC     START
         PUSH 0,GR2        レジスタの待避
         PUSH 0,GR3
         ST   GR1,M
         XOR  GR1,GR1      結果の初期化
         LD   GR3,M
LOOP2    AND  GR3,MASK2    ループ回数→GR3
         JZE  NDIGIT
LOOP3    CALL ADDP
         CPA  GR0,=0
         JNZ  FIN3         あふれ
         SUBA GR3,=1
         JNZ  LOOP3
NDIGIT   LD   GR3,M
         SRL  GR3,4        M÷10
         JZE  FIN3
         ST   GR3,M
         CPA  GR2,LIMIT
         JPL  FIN2         あふれ
```

```
        JZE   FIN2
        SLL   GR2,4        N×10
        JUMP  LOOP2
FIN2    LAD   GR0,1
FIN3    POP   GR3          レジスタの復元
        POP   GR2
        RET
MASK2   DC    #000F
LIMIT   DC    #1000
M       DS    1
        END
```

c,eに関する解答群

| | | | | |
|---|---|---|---|---|
| ア 0 | イ 70 | ウ 78 | エ 168 | オ 737 |
| カ 2529 | キ 3528 | ク 3579 | ケ 5535 | コ 7158 |

d,fに関する解答群

| | | | | |
|---|---|---|---|---|
| ア 0 | イ 1 | ウ 2 | エ 3 | オ 6 |
| カ 9 | キ 45 | ク 51 | ケ 123 | コ 3579 |

**解き方**

設問1

BCDコードの意味がわからなくても、問題文にその説明がありますので、この点は問題ないと思われます。この問題を的確に解くキーワードは、汎用レジスタの役割を正しく把握することにあります。プログラムの構造は、それほど難しくありません。データの下位の桁から1桁(4ビット)ずつ取り出して、加算し、桁上がりが発生したときだけ、それを次の桁に加えます。そこで、副プログラムADDPをよく読んで、最終的な答えがGR1に、最後にけた上げがあったかどうかをGR0にセットしていることを見抜く必要があります。そうすれば、GR1＝$5298_{(16)}$、GR2＝$4853_{(16)}$であるので、その和は10151ですから、GR1＝$0151_{(16)}$,GR0＝1となります。aはイ、bはウとなります。

設問2

この設問では、副プログラム(CALC)が何をしているのか明示されていません。そこで、プログラムを追いかけることになります。与えられたデータ(たとえば、m＝123、n＝45など)を想定して、トレースを試みます。ところで、この問題では、設問2の最後に「表中および解答群中の値はすべて10進数とする」と記述してあります。この記述は、多少あいまいですが、本問題の意図はBCDコードの設問です。最近の情報処理技術者試験では、受験者をトリッキーにひっかけようという問題はまずないと考えられるので、ここは素直に、m＝123ということはBCDコードで$0123_{(16)}$を意味していると考えましょう。

さて、プログラムをトレースするといっても、そのすべてを逐一トレースしていったのでは、時間がかかって現実的ではありません。特に、ループには気を付ける必要があります。場合によっては、数百回から数千回もループを回ることもあります。ループの場合には、それを制御するカウンタとな

る汎用レジスタ(記憶領域をカウンタとすることもできますが、効率の面からたいては汎用レジスタを使います)をチェックし、さらにその継続条件あるいは脱出条件などをチェックする必要があります。このプログラムでは、コメントにあるようにラベルLOOP2のところでGR3をカウンタに使っていることが分かります。この値は、Mの下4ビット、すなわち、BCDコードの最下位桁です。そのループで何をしているか調べると、その中にさらにループ(LOOP3～JZN LOOP3)があって、実はこれがGR3の値の回数だけループしています。そのループでは、副プログラム ADDP を呼び出して、「GR1←GR1+GR2,GR0←桁上がり情報」をセットしています。この計算に先立ち、GR1はゼロになっていますから、Mの一番下の桁の数、すなわち3回、GR2(＝n)の値を加えていることが分かります。そして、さらに調べると、Mを10で割って(M÷10、BCDコードを4ビット右へシフトするということは1桁ずれる、すなわち10で割ることを意味する)、Nの値を10倍(N×10、BCDコードを4ビット左へシフト)していますので、次は、10×NをMの2桁目(0123なら2回)加えていることが分かります。結局、我々が手で計算するのと同じ方式で乗算処理をしていることが分かります。ただし、コメントにもあるように「あふれ」処理をしていることに注意します。

m＝123, n＝45のときは、m×n＝5535で、「あふれ」は起きていません。そこで、cはケとなります。

ADDPは、mの各桁分だけ呼ばれます(45を3回加えて、450を2回加えて、4500を1回加える)ので、dはオとなります。m＝51, n＝3579は、m×n＝182529ですから4桁ではオーバーフローします。これがどのタイミングで見つかるかで答えが決まります。まず、最初に、GR1に3579を1回加える処理をしますが、これはオーバーフローしません。次に、mの値を10で割って5を求めます。そして、3579を10倍しても可能かどうかチェックするために、3579と1000(ともにBCDコード)を比較します(CPA GR2,LIMIT)。ところが、3579は1000より大きいので、「JPZ FIN2」によって終了してしまいます。すると、GR1には3579が残り、ADDPが呼ばれた回数は1となります。したがって、eはク、fはイとなります。

解答

a イ　b ウ　c ケ　d オ　e ク　f イ

# 付録　アセンブラ言語の仕様

## 1．システムCOMETIIの仕様

### 1.1　ハードウェアの仕様

(1)　1語は16ビットで、そのビット構成は、次のとおりである。

| 上位8ビット | | | | | | | | 下位8ビット | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | (ビット番号) |

15：符号（負：1、非負：0）

(2)　主記憶の容量は65536語で、そのアドレスは0～65535番地である。
(3)　数値は、16ビットの2進数で表現する。負数は、2の補数で表現する。
(4)　制御方式は逐次制御で、命令語は1語長又は2語長である。
(5)　レジスタとして、GR(16ビット)、SP(16ビット)、PR(16ビット)、FR(3ビット)の4種類がある。

GR(汎用レジスタ、General Register)は、GR0～GR7の8個があり、算術、論理、比較、シフトなどの演算に用いる。このうち、GR1～GR7のレジスタは、指標レジスタ(index register)としてアドレスの修飾にも用いる。

SP(スタックポインタ、Stack Pointer)は、スタックの最上段のアドレスを保持している。

PR(プログラムレジスタ、Program Register)は、次に実行すべき命令語の先頭アドレスを保持している。

FR(フラグレジスタ、Flag Register)は、OF(Overflow Flag)、SF(Sign Flag)、ZF(Zero Flag)と呼ぶ3個のビットからなり、演算命令などの実行によって次の値が設定される。これらの値は、条件付き分岐命令で参照される。

| | |
|---|---|
| OF | 算術演算命令の場合は、演算結果が－32768～32767に収まらなくなったとき1になり、それ以外のとき0になる。論理演算命令の場合は、演算結果が0～65535に収まらなくなったとき1になり、それ以外のとき0になる。 |
| SF | 演算結果の符号が負(ビット番号15が1)のとき1、それ以外のとき0になる。 |
| ZF | 演算結果が零(全部のビットが0)のとき1、それ以外のとき0になる。 |

(6)　論理加算又は論理減算は、被演算データを符号のない数値とみなして、加算又は減算する。

### 1.2　命令

命令形式及びその機能を示す。ここで、一つの命令コードに対し2種類のオペランドがある場合、上段はレジスタ間の命令、下段はレジスタと主記憶間の命令を表す。

| 命　令 | 書き方 | | 命令の説明 | FRの設定 |
|---|---|---|---|---|
| | 命令コード | オペランド | | |

(1)　ロード、ストア、ロードアドレス命令

| | | | | |
|---|---|---|---|---|
| ロード<br>LoaD | LD | r1,r2 | r1 ← (r2) | ○*1 |
| | | r,adr [,x] | r ← (実効アドレス) | |
| ストア<br>STore | ST | r,adr [,x] | 実効アドレス ← (r) | — |
| ロードアドレス<br>Load ADress | LAD | r,adr [,x] | r ← 実効アドレス | |

(2)　算術、論理演算命令

| 命令 | ニーモニック | オペランド | 動作 | FR |
|---|---|---|---|---|
| 算術加算<br>ADD Arithmetic | ADDA | r1,r2 | r1 ← (r1)＋(r2) | ○ |
| | | r,adr [,x] | r ← (r)＋(実効アドレス) | |
| 論理加算<br>ADD Logical | ADDL | r1,r2 | r1 ← (r1)＋L(r2) | |
| | | r,adr [,x] | r ← (r)＋L(実効アドレス) | |
| 算術減算<br>SUBtract Arithmetic | SUBA | r1,r2 | r1 ← (r1)－(r2) | |
| | | r,adr [,x] | r ← (r)－(実効アドレス) | |
| 論理減算<br>SUBtract Logical | SUBL | r1,r2 | r1 ← (r1)－L(r2) | |
| | | r,adr [,x] | r ← (r)－L(実効アドレス) | |
| 論理積<br>AND | AND | r1,r2 | r1 ← (r1)AND(r2) | ○*1 |
| | | r,adr [,x] | r ← (r)AND(実効アドレス) | |
| 論理和<br>OR | OR | r1,r2 | r1 ← (r1)OR(r2) | |
| | | r,adr [,x] | r ← (r)OR(実効アドレス) | |
| 排他的論理和<br>exclusive OR | XOR | r1,r2 | r1 ← (r1)XOR(r2) | |
| | | r,adr [,x] | r ← (r)XOR(実効アドレス) | |

(3)　比較演算命令

| 命令 | ニーモニック | オペランド | 動作 | FR |
|---|---|---|---|---|
| 算術比較<br>ComPare Arithmetic | CPA | r1,r2 | (r1)と(r2)、又は(r)と(実効アドレス)の算術比較又は論理比較を行い、比較結果によって、FRに次の値を設定する。 | ○*1 |
| | | r,adr [,x] | | |
| 論理比較<br>ComPare Logical | CPL | r1,r2 | | |
| | | r,adr [,x] | | |

| 比較結果 | FRの値 SF | ZF |
|---|---|---|
| (r1)＞(r2)<br>(r)＞(実効アドレス) | 0 | 0 |
| (r1)=(r2)<br>(r)=(実効アドレス) | 0 | 1 |
| (r1)＜(r2)<br>(r)＜(実効アドレス) | 1 | 0 |

(4)　シフト演算命令

| 命令 | ニーモニック | オペランド | 動作 | FR |
|---|---|---|---|---|
| 算術左シフト<br>Shift Left Arithmetic | SLA | r,adr [,x] | 符号を除き(r)を実効アドレスで指定したビット数だけ左又は右にシフトする。<br>シフトの結果、空いたビット位置には、左シフトのときは0、右シフトのときは符号と同じものが入る。 | ○*2 |
| 算術右シフト<br>Shift Right Arithmetic | SRA | r,adr [,x] | | |
| 論理左シフト<br>Shift Left Logical | SLL | r,adr [,x] | 符号を含み(r)を実効アドレスで指定したビット数だけ左又は右にシフトする。<br>シフトの結果、空いたビット位置には、0が入る。 | |
| 論理右シフト<br>Shift Right Logical | SRL | r,adr [,x] | | |

(5)　分岐命令

| 命令 | ニーモニック | オペランド | 動作 | FR |
|---|---|---|---|---|
| 正分岐<br>Jump on PLus | JPL | adr [,x] | FRの値によって、実効アドレスに分岐する。分岐しないときは、次の命令に進む。 | — |
| 負分岐<br>Jump on MInus | JMI | adr [,x] | | |
| 非零分岐<br>Jump on Non Zero | JNZ | adr [,x] | | |
| 零分岐<br>Jump on ZEro | JZE | adr [,x] | | |
| オーバーフロー分岐<br>Jump on OVerflow | JOV | adr [,x] | | |
| 無条件分岐<br>unconditional JUMP | JUMP | adr [,x] | 無条件に実効アドレスに分岐する。 | |

| | FR OF | SF | ZF |
|---|---|---|---|
| JPL | | 0 | 0 |
| JMI | | 1 | |
| JNZ | | | 0 |
| JNE | | | 1 |
| JOV | 1 | | |

(6)　スタック操作命令

| 命令 | ニーモニック | オペランド | 動作 | FR |
|---|---|---|---|---|
| プッシュ<br>PUSH | PUSH | adr [,x] | SP ← (SP)－L1,<br>(SP) ← 実効アドレス | — |
| ポップ<br>POP | POP | r | r ← ((SP)),<br>SP ← (SP)＋L1 | |

(7)　コール、リターン命令

| 命令 | 書き方 | 命令の説明 | FRの設定 |
|---|---|---|---|
| コール<br>CALL subroutine | CALL　adr [,x] | SP ← (SP)−L1,<br>(SP) ← (PR),<br>PR ← 実効アドレス | — |
| リターン<br>RETurn from subroutine | RET | PR ← ((SP)),<br>SP ← (SP)+L1 | |

(8)　その他

| 命令 | 書き方 | 命令の説明 | FRの設定 |
|---|---|---|---|
| スーパバイザコール<br>SuperVisor Call | SVC　adr [,x] | 実効アドレスを引数として割出しを行う。<br>実行後のGRとFRは不定となる。 | — |
| ノーオペレーション<br>No OPeration | NOP | 何もしない。 | |

（注）
- r,r1,r2　　いずれもGRを示す。指定できるGRはGR0〜GR7
- adr　　アドレスを示す。指定できる値の範囲は0〜65535
- x　　指標レジスタとして用いるGRを示す。指定できるGRはGR1〜GR7
- [ 　]　　[ 　]内の指定は省略できることを示す。
- ( 　)　　( 　)内のレジスタ又はアドレスに格納されている内容を示す。
- 実効アドレス　　adrとxの内容との論理加算値又はその値が示す番地
- ←　　演算結果を、左辺のレジスタ又はアドレスに格納することを示す。
- +L,−L　　論理加算、論理減算を示す。
- FRの設定　　○　：設定されることを示す。
  - ○*1　：設定されることを示す。ただし、OFには0が設定される。
  - ○*2　：設定されることを示す。ただし、OFにはレジスタから最後に送り出されたビットの値が設定される。
  - —　：実行前の値が保持されることを示す。

### 1.3　文字の符号表

(1)　JIS X 0201ラテン文字・片仮名用8ビット符号で規定する文字の符号表を使用する。

(2)　右に符号表の一部を示す。1文字は8ビットからなり、上位4ビットを列で、下位4ビットを行で示す。例えば、間隔、4、H、¥のビット構成は、16進表示で、それぞれ20、34、48、5Cである。16進表示で、ビット構成が21〜7E(及び表では省略しているA1〜DF)に対応する文字を図形文字という。図形文字は、表示(印刷)装置で、文字として表示(印字)できる。

(3)　この表にない文字とそのビット構成が必要な場合は、問題中で与える。

| 行＼列 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|
| 0 | 間隔 | 0 | @ | P | ` | p |
| 1 | ! | 1 | A | Q | a | q |
| 2 | " | 2 | B | R | b | r |
| 3 | # | 3 | C | S | c | s |
| 4 | $ | 4 | D | T | d | t |
| 5 | % | 5 | E | U | e | u |
| 6 | & | 6 | F | V | f | v |
| 7 | ' | 7 | G | W | g | w |
| 8 | ( | 8 | H | X | h | x |
| 9 | ) | 9 | I | Y | i | y |
| 10 | * | : | J | Z | j | z |
| 11 | + | ; | K | [ | k | { |
| 12 | , | < | L | ¥ | l | \| |
| 13 | - | = | M | ] | m | } |
| 14 | . | > | N | ^ | n | ~ |
| 15 | / | ? | O | _ | o | |

## 2. アセンブラ言語CASLIIの仕様

### 2.1 言語の仕様

(1) CASL IIは、COMET IIのためのアセンブラ言語である。
(2) プログラムは、命令行及び注釈行からなる。
(3) 1命令は1命令行で記述し、次の行へ継続できない。
(4) 命令行及び注釈行は、次に示す記述の形式で、行の1文字目から記述する。

| 行の種類 | | 記述の形式 |
|---|---|---|
| 命令行 | オペランドあり | [ラベル] {空白} {命令コード} {空白} {オペランド} [ {空白} [コメント]] |
| | オペランドなし | [ラベル] {空白} {命令コード} [ {空白} [ {;} [コメント]]] |
| 注釈行 | | [空白] {;} [コメント] |

（注）
- [ ]　[ ]内の指定が省略できることを示す。
- { }　{ } 内の指定が必須であることを示す。
- ラベル　その命令の(先頭の語)のアドレスを他の命令やプログラムから参照するための名前である。長さは1～8文字で、先頭の文字は英大文字でなければならない。以降の文字は、英大文字又は数字のいずれでもよい。なお、予約語であるGR0～GR7は、使用できない。
- 空白　1文字以上の間隔文字の列である。
- 命令コード　命令ごとに記述の形式が定義されている。
- オペランド　命令ごとに記述の形式が定義されている。
- コメント　覚え書きなどの任意の情報であり、処理系で許す任意の文字を書くことができる。

### 2.2 命令の種類

命令は、4種類のアセンブラ命令(START,END,DS,DC)、4種類のマクロ命令(IN,OUT,RPUSH,RPOP)及び機械語命令(COMET IIの命令)からなる。その仕様を次に示す。

| 命令の種類 | ラベル | 命令コード | オペランド | 機能 |
|---|---|---|---|---|
| アセンブラ命令 | ラベル | START | [実行開始番地] | プログラムの先頭を定義<br>プログラムの実行開始番地を定義<br>他のプログラムで参照する入口名を定義 |
| | | END | | プログラムの終わりを明示 |
| | [ラベル] | DS | 語数 | 領域を確保 |
| | [ラベル] | DC | 定数[,定数]… | 定数を定義 |
| マクロ命令 | [ラベル] | IN | 入力領域,入力文字長領域 | 入力装置から文字データを入力 |
| | [ラベル] | OUT | 出力領域,出力文字長領域 | 出力装置へ文字データを出力 |
| | [ラベル] | RPUSH | | GRの内容をスタックに格納 |
| | [ラベル] | RPOP | | スタックの内容をGRに格納 |
| 機械語命令 | [ラベル] | | （「1.2　命令」を参照） | |

### 2.3 アセンブラ命令

アセンブラ命令は、アセンブラの制御などを行う。

(1) | START | [実行開始番地] |

START命令は、プログラムの先頭を定義する。

実行開始番地は、そのプログラム内で定義されたラベルで指定する。指定がある場合はその番地から、省略した場合はSTART命令の次の命令から、実行を開始する。

また、この命令につけられたラベルは、他のプログラムから入口名として参照できる。

(2) | END | |

END命令は、プログラムの終わりを定義する。

(3) | DS | 語数 |

DS命令は、指定した語数の領域を確保する。

語数は、10進定数(≧0)で指定する。語数を0とした場合、領域は確保しないが、ラベルは有効である。

(4) | DC | 定数[,定数]… |

DC命令は、定数で指定したデータを(連続する)語に格納する。

定数には、10進定数、16進定数、文字定数、アドレス定数の4種類がある。

| 定数の種類 | 書き方 | 命　令　の　説　明 |
|---|---|---|
| 10進定数 | n | nで指定した10進数値を、1語の2進数データとして格納する。ただし、nが－32768～32767の範囲にないときは、その下位16ビットを格納する。 |
| 16進定数 | #h | hは4けたの16進数(16進数字は0～9、A～F)とする。hで指定した16進数値を1語の2進数データとして格納する(0000≦h≦FFFF)。 |
| 文字定数 | '文字列' | 文字列の文字数(＞0)分の連続する領域を確保し、最初の文字は第1語の下位8ビットに、2番目の文字は第2語の下位8ビットに、…と順次文字データとして格納する。各語の上位8ビットには0のビットが入る。<br>文字列には、間隔及び任意の図形文字を書くことができる。ただし、アポストロフィ(')は2個続けて書く。 |
| アドレス定数 | ラベル | ラベルに対応するアドレスを1語の2進数データとして格納する。 |

### 2.4　マクロ命令

マクロ命令は、あらかじめ定義された命令群とオペランドの情報によって、目的の機能を果たす命令群を生成する(語数は不定)。

(1)

| IN | 入力領域、入力文字長領域 |
|---|---|

IN命令は、あらかじめ割り当てた入力装置から、1レコードの文字データを読み込む。

入力領域は、256語長の作業域のラベルであり、この領域の先頭から、1文字を1語に対応させて順次入力される。レコードの区切り符号(キーボード入力の復帰符号など)は、格納しない。格納の形式は、DC命令の文字定数と同じである。入力データが256文字に満たない場合、入力領域の残りの部分は実行前のデータを保持する。入力データが256文字を超える場合、以降の文字は無視される。

入力文字長領域は、1語長の領域のラベルであり、入力された文字の長さ(≧0)が2進数で格納される。ファイルの終わり(end of file)を検出した場合は、－1が格納される。

IN命令を実行すると、GRの内容は保存されるが、FRの内容は不定となる。

(2)

| OUT | 出力領域、出力文字長領域 |
|---|---|

OUT命令は、あらかじめ割り当てた出力装置に、文字データを、1レコードとして書き出す。

出力領域は、出力しようとするデータが1文字1語で格納されている領域のラベルである。格納の形式は、DC命令の文字定数と同じであるが、上位8ビットは、OSが無視するので0でなくてもよい。

出力文字長領域は、1語長の領域のラベルであり、出力しようとする文字の長さ(≧0)を2進数で格納しておく。

OUT命令を実行すると、GRの内容は保存されるが、FRの内容は不定となる。

(3)

| RPUSH | |
|---|---|

RPUSH命令は、GRの内容を、GR1、GR2、…、GR7の順でスタックに格納する。

(4)

| RPOP | |
|---|---|

RPOP命令は、スタックの内容を順次取り出し、GR7、GR6、…、GR1、の順でGRに格納する。

### 2.5　機械語命令

機械語命令のオペランドは、次の形式で記述する。

r,r1,r2　　GRは、記号GR0～GR7で記述する。

x　　指標レジスタとして用いるGRは、記号GR1～GR7で指定する。

adr　　アドレスは、10進定数、16進定数、アドレス定数又はリテラルで指定する。
リテラルは、一つの10進定数、16進定数又は文字定数の前に等号(＝)を付けて記述する。CASL IIは、等号の後の定数をオペランドとするDC命令を生成し、そのアドレスをadrの値とする。

### 2.6　その他

(1) アセンブラによって生成される命令語や領域の相対位置は、アセンブラ言語での記述順序とする。ただし、リテラルから生成されるDC命令は、END命令の直前にまとめて配置される。

(2) 生成された命令語、領域は、主記憶上で連続した領域を占める。

## 3. プログラム実行の手引

### 3.1　OS

プログラムの実行に関して、次の取決めがある。

(1) アセンブラは、未定義ラベル(オペランド欄に記述されたラベルのうち、そのプログラム内で定義されていないラベル)を、他のプログラムの入口名(START命令のラベル)と解釈する。この場合、アセンブラはアドレスの決定を保留し、その決定をOSに任せる。OSは、実行に先立って他のプログラムの入口名との連係処理を行いアドレスを決定する(プログラムの連係)。

(2) プログラムは、OSによって起動される。プログラムがロードされる主記憶の領域は不定とするが、プログラム中のラベルに対応するアドレス値は、OSによって実アドレスに補正されるものとする。

(3) プログラムの起動時に、OSはプログラム用に十分な容量のスタック領域を確保し、その最後のアドレスに1を加算した値をSPに設定する。
(4) OSは、CALL命令でプログラムに制御を渡す。プログラムを終了しOSに制御を戻すときは、RET命令を使用する。
(5) IN命令に対応する入力装置、OUT命令に対応する出力装置の割当ては、プログラムの実行に先立って利用者が行う。
(6) OSは、入出力装置や媒体による入出力手続の違いを吸収し、システムでの標準の形式及び手続(異常処理を含む)で入出力を行う。したがって、IN、OUT命令では、入出力装置の違いを意識する必要はない。

### 3.2 未定義事項

プログラムの実行等に関し、この仕様で定義しない事項は、処理系によるものとする。

# 参考資料　命令語の構成

| 第1語 | | | | 第2語 | 命令語長 | 命令語とアセンブラとの対応 | |
|---|---|---|---|---|---|---|---|
| op | | r/r1 | x/r2 | adr | | 機械語命令 | 意味 |
| 主OP | 副OP | | | | | | |
| 0 | 0 | — | — | — | 1 | NOP | no operation |
| 1 | 0 | | | | 2 | LD r,adr,x | load |
| | 1 | | | | 2 | ST r,adr,x | store |
| | 2 | | | | 2 | LAD r,adr,x | load address |
| | 4 | | | — | 1 | LD r1,r2 | load |
| 2 | 0 | | | | 2 | ADDA r,adr,x | add arithmetic |
| | 1 | | | | 2 | SUBA r,adr,x | subtract arithmetic |
| | 2 | | | | 2 | ADDL r,adr,x | add logical |
| | 3 | | | | 2 | SUBL r,adr,x | subtract logical |
| | 4 | | | — | 1 | ADDA r1,r2 | add arithmetic |
| | 5 | | | — | 1 | SUBA r1,r2 | subtract arithmetic |
| | 6 | | | — | 1 | ADDL r1,r2 | add logical |
| | 7 | | | — | 1 | SUBL r1,r2 | subtract logical |
| 3 | 0 | | | | 2 | AND r,adr,x | and |
| | 1 | | | | 2 | OR r,adr,x | or |
| | 2 | | | | 2 | XOR r,adr,x | exclusive or |
| | 4 | | | — | 1 | AND r1,r2 | and |
| | 5 | | | — | 1 | OR r1,r2 | or |
| | 6 | | | — | 1 | XOR r1,r2 | exclusive or |
| 4 | 0 | | | | 2 | CPA r,adr,x | compare arithmetic |
| | 1 | | | | 2 | CPL r,adr,x | compare logical |
| | 4 | | | — | 1 | CPA r1,r2 | compare arithmetic |
| | 5 | | | — | 1 | CPL r1,r2 | compare logical |
| 5 | 0 | | | | 2 | SLA r,adr,x | shift left arithmetic |
| | 1 | | | | 2 | SRA r,adr,x | shift right arithmetic |
| | 2 | | | | 2 | SLL r,adr,x | shift left logical |
| | 3 | | | | 2 | SRL r,adr,x | shift right logical |
| 6 | 1 | — | | | 2 | JMI adr,x | jump on minus |
| | 2 | — | | | 2 | JNZ adr,x | jump on non zero |
| | 3 | — | | | 2 | JZE adr,x | jump on zero |
| | 4 | — | | | 2 | JUMP adr,x | unconditional jump |
| | 5 | — | | | 2 | JPL adr,x | jump on plus |
| | 6 | — | | | 2 | JOV adr,x | jump on overflow |
| 7 | 0 | — | | | 2 | PUSH adr,x | push |
| | 1 | | — | — | 1 | POP r | pop |
| 8 | 0 | — | | | 2 | CALL adr,x | call subroutine |
| | 1 | — | — | — | 1 | RET | return from subroutine |
| 9<br>～<br>E | | | | | | その他の命令 | |
| F | 0 | — | | | 2 | SVC adr,x | supervisor call |

（出題範囲：p.21の表より引用）

# 索　引

（数字・アルファベット）

## 数字



## A



## B



## C



## D



## E



## F



## H



## I



## J



## K



## L

## (五十音順)

### あ



### か



### さ



### た

## な



## は



## ま



## や



## ら



## わ

## 著者紹介


■内 田 智 史（うちだ さとし）

【学歴】
昭和57年　青山学院大学理工学部経営工学科卒業
昭和62年　青山学院大学大学院博士後期課程単位取得済み退学
平成７年　博士（工学）筑波大学

【職歴】
青山学院大学理工学部助手、神奈川大学工学部助手、専任講師を経て
平成11年　神奈川大学工学部経営工学科助教授、現在に至る。


【著書】
・間野浩太郎監修・内田智史著『TURBO Pascalプログラミング』オーム社、1990.6
・ASNOP研究会編(共著)『非線形最適化プログラミング』日刊工業新聞社、1991.4
・内田智史編著『C言語によるプログラミング[基礎編]』オーム社、1991.11
・内田智史編著『C言語によるプログラミング[応用編]』オーム社、1992.2
・内田智史・西浦康友・角田敏雄共著『ソフトウェアの基礎』マグロウヒル、1994.4
・内田智史・秋元勝・北川雅巳・大津崇共著
『C言語によるプログラミング[スーパーリファレンス編]』オーム社、2000.1
・内田智史著『理工系のためのC言語プログラミング入門』科学技術出版、2000.5
・内田智史監修・システム計画研究所編『C言語によるプログラミング[基礎編]第2版』
オーム社、2001.11

---

＊本書および付属CD-ROMの使用により、万一損害が発生しても一切の責任を負いかねますので、あらかじめご了承下さい。
＊本書および付属CD-ROMに対するご質問等につきましては、お手紙にて下記発行所あてご連絡下さい。なお、お電話や電子メールによるご質問等につきましてはお受けできかねますので、併せてご了承下さい。


〒164-0001
東京都中野区中野5丁目62番地1号（EDCビル）
**電子開発学園出版局**

情報処理基礎講座
アセンブラ入門（CASL Ⅱ）
ISBN4-88647-607-4

2001年 4 月 1 日　初版第１刷発行
2006年 10 月 1 日　２版第５刷発行

著　者／内田智史
発行人／松尾　泰
発行所／電子開発学園出版局
発売元／株式会社エスシーシー（SCC）
〒164-8505
東京都中野区中野5丁目62番1号（EDCビル）
電話（03）3319-7101
印刷・製本所／株式会社カントー
電話（03）3238-6011

# アセンブラ入門(CASLⅡ)

# 別冊

## 総合問題・演習問題　解答

電子開発学園出版局

**総合問題1**

**[演習1-1]**

(1) ○、　(2) ×：機械語はそれぞれのコンピュータごとに異なる。JIS規格も存在しない。　(3) ×：アセンブラ言語で記述したプログラムは実行効率は確かによいが、プログラムの生産性が極度に低くなり、プログラムの生産コストを上昇させる。したがって、すべてをアセンブラ言語で開発することは得策ではない。　(4) ○、　(5) ○、　(6) △：基本的にアセンブラ言語は、そのコンピュータのすべての機械語を表現できる。しかし、機械語の中には特権命令と呼ばれる特殊な命令があり、この命令は特権を有したプログラムでないと実行できない。ただし、COMET IIは単純化されたコンピュータなので特権命令はない。　(7) ○、　(8) ×：現在のコンピュータでもプログラム内蔵方式が採用されている。　(9) ×：アセンブラ言語で書かれたプログラムはそのコンピュータの機械語に1対1に変換されるので、実行効率は低下しない。　(10) ×：C言語特有の処理のために、特定のコンピュータに対するチューニングができないので、「常に効率が良くなるとは言えない」。ただし、現在のC言語コンパイラは大変良くできており、へたな人が作成したアセンブラ言語のプログラムよりも効率の良い機械語を生成する。　(11) ×：Windows 98は、パソコンメーカが異なってもインテル社製のCPUを対象としたOSであるので、機械語はまったく同じである。したがってメーカが異なっても同じ機械語が動作する。　(12) ×：CPUが同じであれば機械語は同じである。しかし、OSが異なるので動作環境や実行時ライブラリ(入出力やGUIのプログラム)が異なる。実行時ライブラリが異なれば残念ながら実行できない。

**演習問題2-2**

時間の表記では、時刻に12進法、分・秒に60進法を用いている。また、完全な例ではないが、月は12進法、日にちは変則的な30進法を用いていると言えないこともない。さらに、算盤(そろばん)は、1~4までの珠と5の珠の2種類の珠を用いている。これも変則的な5進法と言えないこともない。

**演習問題2-3-1**

(1) (a) 10,　(b) 59,　(c) 171

(2)

| 数値 | 10進数 | 「奇数」、 | 「2の倍数(=偶数)」、 | 「4の倍数」、 | 「8の倍数」 |
|---|---|---|---|---|---|
| 11011 | 27 | ○ | × | × | × |
| 100000 | 32 | × | ○ | ○ | ○ |
| 111110 | 62 | × | ○ | × | × |
| 110100 | 52 | × | ○ | ○ | × |
| 100011 | 35 | ○ | × | × | × |

**演習問題2-3-2**

(1) $11000_{(2)} = 24_{(10)}$,　(2) $1000000_{(2)} = 64_{(10)}$

**演習問題2-4**

(1) 11,　(2) 47,　(3) 668

**演習問題2-5**

(1) 101110011001010,　(2) 1111110111011010,　(3) 1001000110100,　(4)100010001000100,　(5) 1100101011111110

**演習問題2-6-1**

(1) 符号付き：−1, 符号無し:15、　(2) 符号付き：−6, 符号無し:10、　(3) 符号付き：−2, 符号無し:14、　(4) 符号付き：1, 符号無し:1、　(5) 符号付き：5, 符号無し:5

**演習問題2-6-2**

(1) 1111111111111011,　(2) 1111111111110110,　(3) 0000000000000101

**演習問題2-6-3**

(1) 0000000000000101 ＋ 1111111111111101 = 0000000000000010

(2) 0000000000000010 ＋ 1111111111110110 = 1111111111111000

(3) 0000000000001110 ＋ 1111111111111011 = 0000000000001001

**演習問題2-7-1(1)**

(1) 10011111,　(2) 100001101000,　(3) 10011100010000

**演習問題2-7-1(2)**

(1) 237,　(2) 4150,　(3) 23420

**演習問題2-7-1(3)**

(1) 9F,　(2) 868,　(3) 2710

**演習問題2-7-2**

(1) $135_{(8)} = 5D_{(16)}$,　(2) $3607417_{(8)} = F0F0F_{(16)}$,　(3) $255325_{(8)} = 15AD5_{(16)}$

**演習問題2-7-3**

(1) 1101110,　(2) 1111111110101011

**演習問題2-7-4**

(1) 29C(16),　(2) 125715(8),　(3) FFF(16)

**演習問題2-7-5**

(1) 427,　(2) 5349,　(3) 252

**総合問題2**

**[演習2-1]**

(a) (1) 101,　(2) 1010,　(3) 101101,　(4) 1010001010,　(5) 10000000001

(b) (1) 7(8),　(2) 175(8),　(3) 6405(8),　(4) 23613(8),　(5) 303240(8)

(c) (1) 64(16),　(2) 929(16),　(3) 401(16),　(4) FA00(16),　(5) 186A0(16)

**[演習2-2]**

(1) 725,　(2) 470,　(3) 8240817,　(4) 926118497,　(5) 51966,　(6) 19645263

**[演習2-3]**

(1) 1011100(2),　(2) 101011001(2),　(3) 372200(8),　(4) 323713(8),　(5) AB013(16),　(6) FD01FFF(16),　(7) 12ABD(16),
(8) 1000000(16)

**[演習2-4]**

(1) 1001100(2),　(2) 100011(2),　(3) 11101011(2)

**[演習2-5]**

80000を2進数で表現すると、10011100010000000(2)となる。ちなみに16進数では13880(16)である。この下位16ビットは、16進数で4桁に相当するから3880(16)=11100010000000(2)である。この値は、10進数では14464である。したがって、答は14464。

**演習問題3-1**

(1) 各汎用レジスタは16ビットであるから、符号付きの場合−32768～32767まで表現でき、符号無しの場合0～65535まで表現できる(表2-9参照)。
(2) PRは16ビットなので、0～65535までの数を表現できる。したがって、COMET IIの記憶装置のすべての番地をPRで表現できる。

**演習問題3-2**

(1) LD GR5,7936 あるいは LD GR5,#1F00 (注：7936=1F00(16))
(2) 101003E8 (注：1000=03E8(16))

**演習問題3-3**

(1) PRは次に実行すべき機械語の格納された番地を保持している。つまり、実行中の機械語の命令が終了すると、PRの番地の機械語命令がCPUに取り込まれ解読される。その時に、PRの値は自動的に機械語の語長分だけ増加される。つまり、実行中の機械語命令が1語長であれば、PRは現在実行中の機械語の格納された番地+1になっている。語長が2であれば、PRは現在実行中の機械語の格納された番地+2になっている。ただし、機械語の種類(ジャンプ命令、CALL命令、RET命令)によってはPRを書き直すので注意すること。
(2) ロードして実行するプログラムが4000バイト=4000語である。もし、61536番地からロードすると、プログラムは61536番地～65535番地に格納される。しかし、61537番地以降からロードすると、プログラムの最後の部分が格納できなくなる。したがって、0番地から61536番地の間にロードすることができる。ちなみに61536=65535−4000+1である。
(3) (a) PRが指している記憶装置のデータを機械語命令として取り出す。そのとき、第1語目を見れば、その語長が分かるので、必要に応じて第2語を取り出す。(b) PRの値を語長分だけ増加させる。(c) 取り出してきた機械語命令を実行する。(d) ステップ(a)に戻る

**演習問題 3-4-2**

```
SAMP1E    START
          LD     GR1,7
          ADDA   GR1,7
          ST     GR1,8
          RET
          DC     5
          DS     1
          END
```

「ADDA　GR1,7」は「SLA　GR1,1」でも良い。

**演習問題 3-4-4**

(1) (必要性1) 絶対番地を使うとプログラムの修正などに伴いアドレスがずれた時に修正が困難である。また、プログラムがロードされるアドレスが固定されてしまう。(必要性2) 人間にとって番号で記憶するよりも名前で記憶した方が覚えやすいし、プログラムも分かりやすくなる。

(2) (a) 数字から始まっている、 (b) アンダーラインは使えない、 (c) ハイフンは使えない、 (d) ギリシア文字は使えない、 (e) ラベルは8文字までしか使えない

**総合問題3**

**[演習3-1]**

CASL II アセンブラは、CASL IIプログラムを入力して、COMET IIの機械語を生成する。COMET IIシミュレータは、その機械語を実行する。

**[演習3-2]**

CASLDVを起動して、右上のペイン(ウィンドウ)にプログラムを入力する。メニューから「ファイル→名前を付けて保存」を選び(あるいは、書込みボタンを押す)、SAMP1E.caslというファイル名で保存する(ファイル名は～.caslであれば何でもよい)。メニューから「アセンブル→アセンブル(CASL完全準拠)」を選ぶ(あるいはアセンブルボタンを押す)。これでアセンブル結果が左上のペインに描画される。メニューから「実行→実行開始」を選ぶ(あるいは、実行ボタンを押す)。これで、プログラムが実行される。画面上の8番地の内容が「000A」になっているはずである。

**[演習3-3]**

(1) ×：フラグレジスタは3ビットである。それ以外は16ビットである。一般にレジスタというとPRやFR、SPも含んだ全てのレジスタを指すことに注意。 (2) ×：GR0は指標レジスタとしては使えない。 (3) ×：ロードしてもそのまま記憶装置に残っている。 (4) ×：1語の命令もある。 (5) ○。 (6) ○。 (7) ○。 (8) ×：アンダーラインは使えない。

**[演習3-4]**

(a) 0、 (b) 65535、 (c) 8、 (d) 3、 (e) 8、 (f) オペレーティングシステム

**演習問題4-1**

(ア) ラベルは1カラム目から書き出す。 (イ) プログラムは大文字で記述する。 (ウ) 命令は2カラム以降から記述する。 (エ) オペランドの間には空白を書かない。 (オ) 1行で記述する。 (カ) オペランドがない場合、コメントは;の後に記述する。 (キ) START命令には必ずラベルを書く。 (ク) END命令にはラベルを書かない。 (ケ) GR0～GR7はラベル名としては使えない。

**演習問題4-2**

(1) 100000を16進数に直すと、186A0である。この下位4バイトは86A0である。実際には、この数値が設定される。これを10進数で見ると、符号付きのデータの場合－31072となり、符号無しデータの場合34464となる。なお、これをチェックするプログラムがOVRCHK.caslに格納されている。

(2) 30000を16進数に直すと、7530である。したがって、CASL IIの16進定数では、#7530と表記される。

(3) －1を16進数に直すと、FFFFである。したがって、CASL IIの16進定数では、#FFFFと表記される。

**演習問題4-3**

(1) ○、(2) ○：ただし、最後の「,GR3」がコメントになるが見た目に分かり難いだけでなく、GR3を指標レジスタと勘違いする恐れがある。(3) ×：RET命令はオペランドをとらない。また、コメントを書く場合には、;で開始する。(4) ○

**演習問題4-5-1**

(1) 最初のプログラムの名前はPRG3で実行開始アドレスは0番地である。次のプログラムの名前はPRG4で実行開始アドレスは3番地である。

(2) START命令に記述する実行開始アドレスはラベルでなければならない。この例では絶対番地を指定している。

**演習問題4-5-2**

END命令はアセンブラに対して、「ここでプログラムが終了しますよ」ということを伝えるための命令であり、これに対応する機械語はない。機械語命令がなければ、それに対するアドレスを決定することができない。

**演習問題4-5-3**

(1) 4語、 (2) 8語、 (3) 3語(アドレス定数は1語分占める)

**演習問題4-5-4**

DS命令のあとに記述できるのは0以上の10進定数だけである。

(1) 文字定数は記述できない、 (2) 16進定数は記述できない、 (3) 負の数は記述できない

**総合問題4**

**[演習4-1]**

以下の説明の中で数値はすべて16進数とする。

START命令は、対応する機械語がないので、何もしない。「LD GR1,A」は、主記憶装置の内容(Aの値)を汎用レジスタに転送する命令であり、この機械語は「10」で始まる。次に汎用レジスタの番号1と(後で学習する指標レジスタを使用しないという意味の)0が続き、第0語目は「1010」となる。次にラベルAのアドレスを記述するが、この時点ではまだAが配置されるアドレスが分からないので第1語目は保留にしておく。「ADDA GR1,B」は、主記憶装置の内容(Bの値)を汎用レジスタに算術加算する命令であり、この機械語は「20」で始まる。次に汎用レジスタの番号1と(後で学習する指標レジスタを使用しないという意味の)0が続き、第2語目は「2010」となる。次にラベルBのアドレスを記述するが、この時点ではまだBが配置されるアドレスが分からないので保留にしておく。「ST GR1,C」は、汎用レジスタの内容を主記憶装置(C)に転送する命令であり、これは「11」で始まる。次に汎用レジスタの番号1と(後で学習する指標レジスタを使用しないという意味の)0となり、第4語目は「1110」となる。次にラベルCのアドレスを記述するが、この時点ではまだCが配置されるアドレスが分からないので保留にしておく。「RET」は1語の命令であり、これは「8100」である。これが第6語目に配置される。次にラベルAの定義があるが、ここで初めてラベルAが第7語目に配置されることがわかる。そこで、最初の命令「LD GR1,A」は「1010 0007」となり、これが第0語目～第1語目となる。また、「A DC 5」は値5がそのままデータとして残るので第7語目は「0005」となる。同様に「B DC 3」は、「0003」となり、これが第8語目に配置される。Bの相対アドレスが8となるので、この時点で「ADDA GR1,B」は「2010 0008」となる。「C DS 1」によってCのアドレスが9であり、ここから1語分が確保されることがわかる。ただし、DS命令なので、第9語目に配置されるデータは不定となる(つまり、何がセットされるかは分からない)。しかし、Cのアドレスは分かったので、「ST GR1,C」は「1110 0009」となる。以上をまとめると、以下(表4-1と同じ)に示す機械語が生成できる。ただし、ここで生成したA, B, Cのアドレスは相対アドレスであり、さらにローダによって実際のアドレスに変換されることに注意されたい。

したがって、以下の機械語が生成される(????は何が入るか分からないことを示す)。

```
1010 0007 2010 0008 1110 0009 8100 0005 0003 ????
```

**[演習4-2]**

(1) ×：ラベルは必ず第1カラムから記述する、 (2) ×：最大8文字まで、 (3) ○：ラベル名として使えないのは、汎用レジスタの名前だけである、 (4) ×：必ずしも付ける必要はない、 (5) ×：必ず付ける、 (6) ×：DS命令には、このような書き方は許されていない、 (7) ○、 (8) ○、 (9) ×：オペランドがある場合には、セミコロン；を付けずに直接コメントを書くことができる、 (10) ×：「DC 'AB'' C'」と書くべきである。

**演習問題5-1-1**

(1)
```
EX511A  START
        LD    GR1,X
        SUBA  GR1,Y
        ADDA  GR1,Z
        ST    GR1,W
        RET
X       DC    10
Y       DC    11
Z       DC    12
W       DS    1
        END
```

(2)
```
EX511B  START
        LD    GR1,X
        SUBA  GR1,Y
        SUBA  GR1,Z
        ST    GR1,W
        RET
X       DC    10
Y       DC    11
Z       DC    12
W       DS    1
        END
```

**演習問題5-1-2**

```
EX512   START
        LD    GR1,A
        ADDA  GR1,B    GR1←A＋B
        LD    GR2,C
        ADDA  GR2,D    GR2←C＋D
        LD    GR3,GR1  GR3←GR1(A＋B)
        ADDA  GR3,GR1  GR3←2(A＋B)
        SUBA  GR3,GR2  GR3←2(A＋B)－(C＋D)
        SUBA  GR3,GR2  GR3←2(A＋B)－2(C＋D)
        SUBA  GR3,GR2  GR3←2(A＋B)－3(C＋D)
        ST    GR3,E
        RET
A       DC    10
B       DC    11
C       DC    12
D       DC    13
E       DS    1
        END
```

**演習問題5-2-1**

たとえば、4ビットのデータで考える。$b_3b_2b_1b_0$という2進数($b_i$=1か0)は$b_3\times2^3+b_2\times2^2+b_1\times2^1+b_0\times2^0$である。この各ビットを左に1ビットシフトするということは$b_3b_2b_1b_00$という2進数になり、これは$b_3\times2^4+b_2\times2^3+b_1\times2^2+b_0\times2^1$という数になり、全体を2で括ると$2\times(b_3\times2^3+b_2\times2^2+b_1\times2^1+b_0\times2^0)$となるので、元の数の2倍になっていることがわかる。また、$b_3b_2b_1b_0$を1ビット右にシフトするということは$b_3b_2b_1$という2進数になり、これは$b_3\times2^2+b_2\times2^1+b_1\times2^0$

という数になり、元の数の半分になっている。

**演習問題5-2-2**

(1)
```
EX522A  START
        LD      GR1,N
        SLA     GR1,2
        ST      GR1,M
        RET
N       DC      10
M       DS      1
        END
```

(2)
```
EX522B  START
        LD      GR1,N
        SLA     GR1,3
        ST      GR1,M
        RET
N       DC      10
M       DS      1
        END
```

(3)
```
EX522C  START
        LD      GR1,A
        ADDA    GR1,B
        SLA     GR1,3
        ST      GR1,C
        RET
A       DC      3
B       DC      7
C       DS      1
        END
```

**演習問題5-2-3**

(1)
```
EX523A  START
        LD      GR1,N
        LD      GR2,GR1    GR2←N
        SLA     GR2,1      GR2←2N
        LD      GR3,GR1    GR3←N
        SLA     GR3,3      GR3←8N
        ADDA    GR1,GR2    GR1←N+2N=3N
        ADDA    GR1,GR3    GR1←3N+8N=11N
        ST      GR1,M
        RET
N       DC      10
M       DS      1
        END
```

(2)
```
EX523B  START
        LD      GR1,A
        ADDA    GR1,B
        LD      GR2,GR1    GR2←(A+B)
        SLA     GR2,1      GR2←2(A+B)
        LD      GR3,GR1    GR3←(A+B)
        SLA     GR3,3      GR3←8(A+B)
        ADDA    GR1,GR2    GR1←(A+B)+2(A+B)=3(A+B)
        ADDA    GR1,GR3    GR1←3(A+B)+8(A+B)=11(A+B)
        ST      GR1,C
        RET
A       DC      3
B       DC      7
C       DS      1
        END
```

**演習問題5-2-4**

(1)
```
EX524A  START
        LD      GR1,N
        SRA     GR1,3
        ST      GR1,M
        RET
N       DC      80
M       DS      1
        END
```

(2)
```
EX524B  START
        LD      GR1,A
        ADDA    GR1,B
        SRA     GR1,4
        ST      GR1,C
        RET
A       DC      80
B       DC      1520
C       DS      1
        END
```

**演習問題5-3-1**

指標レジスタの計算を単純化するためである。もし、表の先頭を1とすると、実効アドレスの計算は「アドレス＋指標レジスタ＋1」となって余分な加算(網掛け部)が必要となる。

**演習問題5-3-2**

(1) 100, (2) 200, (3) 102, (4) 201, (5) 204, (6) 105

ただし、設問(4)と(6)では、指標レジスタの値が目的としているテーブルの範囲を超えている。たまたま、TABLE1とTABLE2が連続して配置されていたので、このような結果となったのであり、本来はこのようなプログラムは書くべきではない。しかし、TABLE1とTABLE2が初めから連続して配置してあり、一つの連続したテーブルであることを想定している場合は、問題はない(が、よほどうまくロジックを組まないとプログラムが分かり難くなる)。

**演習問題5-4**

実行されない。JUMP命令は無条件にジャンプしてしまうため、その直後の命令は実行されない。CASLDVにこのプログラムをかけると警告メッセージが表示される。

**演習問題5-5**

(1) (a) LD GR0,='a'　(b) LD GR1,=−45 (c) LD GR2,#FFFF

(2) 「LD GR2,='CASL II'」は、次のようにCASL IIアセンブラによって変換される。

```
          LD GR2,TEMPDATA
               ↙
TEMPDATA   DC  'C'
           DC  'A'
           DC  'S'
           DC  'L'
           DC  ' '
           DC  'I'
           DC  'I'
```

したがって、GR2には'C'の文字コード$43_{(16)}$が入る。

**総合問題5**

**[演習5-1]**

(1) Xが9番地、YがA番地、ZがB番地、WがC番地、　(2) Xが3F1番地、Yが3F2番地、Zが3F3番地、Wが3F4番地、
(3) Xが2009番地、Yが200A番地、Zが200B番地、Wが200C番地

**[演習5-2]**

へたに(第9章で学習する)ループを使ってプログラムを組むよりも、次のプログラムの方が単純で高速である。

```
ST  GR0,X
ST  GR0,X,GR1
ST  GR0,X,GR2
ST  GR0,X,GR3
ST  GR0,X,GR4
```

**[演習5-3]**

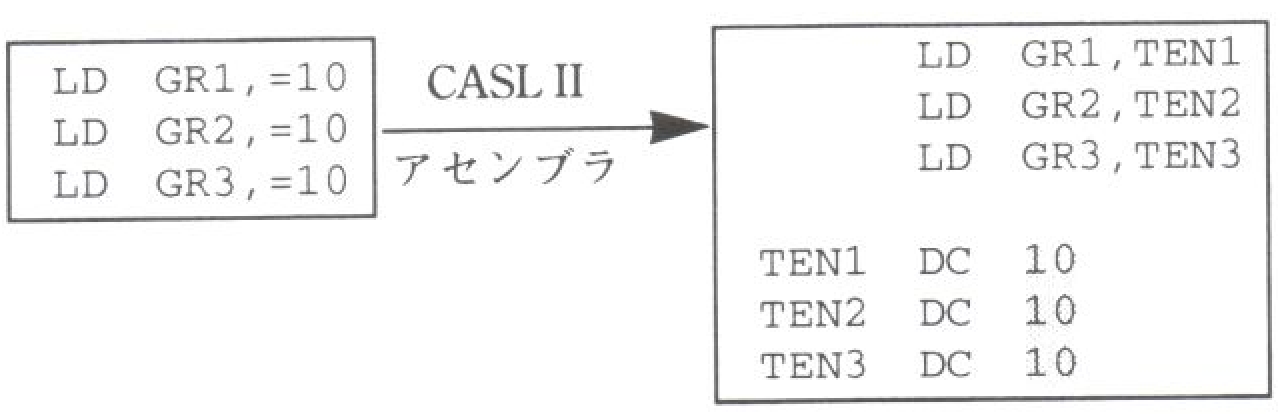
(1) ×：たとえば、レジスタGR0の値を32倍するには「SLA GR0,5」とすれば1命令でよい。　(2) ○。　(3) ○：ただし、「DC 1,2」は2語確保し、そこにデータ1,2をセットするのに対し、「DS 2」は2語データを確保するが、そこに設定される値は不定である。　(4) ×：指標レジスタとして使えるのは、GR1からGR7までの7個の汎用レジスタである。GR0は指標レジスタとしては使えない。　(5) ×：「LD GR1,X,GR1」ができないという記述は、COMET IIの定義のどこにも書かれていない。たとえば、LD命令の例ではないが、実際に、「LAD GR1,1,GR1」という記述はよく用いる。　(6) ×：この記述もどこにも書かれていない。　(7) ○。　(8) ○。　(9) ×：「LD GR1,150」によって生成される実効アドレスは150である。　(10) ○。

**演習問題7-1-1**

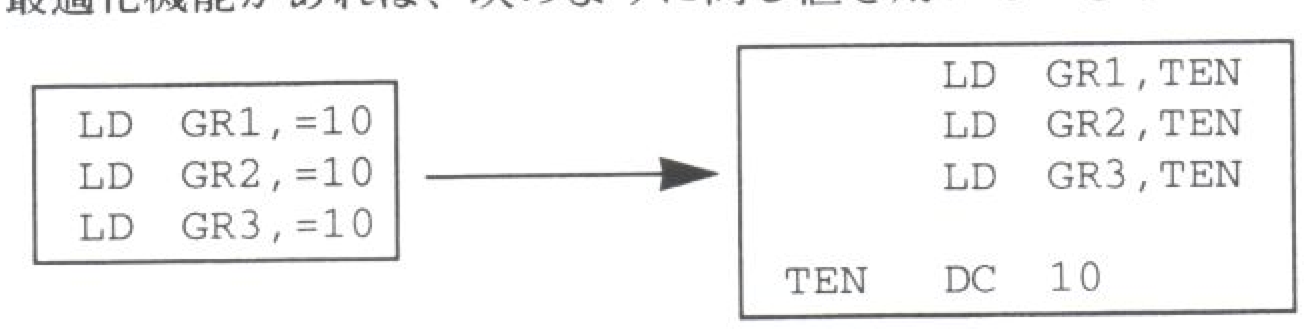
(1) (a) `LAD GR7,12345`　(b)
```
LD GR7,CONST
CONST  DC 12345
```
(c) `LD GR7,=12345`

(2) (方法1)
```
LD  GR0,VALUE
LD  GR1,GR0
LD  GR2,GR0
LD  GR3,GR0
LD  GR4,GR0
LD  GR5,GR0
LD  GR6,GR0
LD  GR7,GR0
```
(方法2)
```
LD  GR0,VALUE
LD  GR1,VALUE
LD  GR2,VALUE
LD  GR3,VALUE
LD  GR4,VALUE
LD  GR5,VALUE
LD  GR6,VALUE
LD  GR7,VALUE
```

方法1では、最初の命令だけが2語であとはすべて1語であるので、プログラム全体の語数は9語である。また、主記憶装置とのアクセスは最初の1命令だけであとは全部CPU内のデータ複写となっている。これに対し、方法2では、すべての命令が2語なので、プログラム全体の語数は16語となっている。さらに、すべての命令で主記憶装置とのアクセスが発生している。このことから方法1の方がかなり実行効率および記憶効率が良いと言える。
(3) プログラム1では3つのLD命令がTENという領域のデータを汎用レジスタに転送している。データが定義されている領域(TENの領域)は1語分しかない。しかし、プログラム2では、それぞれのLD命令で別個にリテラルを用いている。これらの値はすべて10という同じ数であるが、最適化機能がないということから、これらは次に示すように別々の領域に確保される可能性が高い。




このため、重複したデータ(網掛け)が主記憶装置内に存在してしまうことになる。
(4) 最適化機能があれば、次のように同じ値を用いているリテラルを1つの領域に統合することも考えられる。




このようにすれば、データ10のための記憶装置は1語で済む。

**演習問題7-1-2**

(1) (方法1)

```
XOR   GR0,GR0
XOR   GR1,GR1
XOR   GR2,GR2
XOR   GR3,GR3
XOR   GR4,GR4
XOR   GR5,GR5
XOR   GR6,GR6
XOR   GR7,GR7
```

(方法2)

```
XOR   GR0,GR0
LD    GR1,GR0
LD    GR2,GR0
LD    GR3,GR0
LD    GR4,GR0
LD    GR5,GR0
LD    GR6,GR0
LD    GR7,GR0
```

仕様書には詳しく書いていないが、ゼロを作り出すにはXOR命令が最も効率が良いと考えられる。しかし、その後、すでにあるゼロ(GR0)をLD命令で複写する方が効率が良いのか、あるいはXOR命令ですべての汎用レジスタをゼロにすれば良いのかは、仕様書からは読み取れないので、この2つを解として示す。

(2) `AND GR3,=1`

解説：最下位ビットに注目する。偶数なら最下位ビットが0で、奇数なら1である。そこで、1と論理積(AND)を取ればよい。

(3) `XOR GR4,=1`

解説：1と排他的論理和を取ると問題文の処理になる。

**演習問題7-2-1**

(1)

```
LD  GR1,GR0
LD  GR2,GR0
LD  GR3,GR0
LD  GR4,GR0
LD  GR5,GR0
LD  GR6,GR0
LD  GR7,GR0
```

(2) 「LAD GR5,0,GR3」によって生成される実効アドレスは「0＋GR3の値」であるから、GR3の値が実効アドレスになり、この値がそのままGR5に複写される。「LD GR5,0,GR3」の場合、生成される実効アドレスは同じく「0＋GR3の値」である。LD命令の場合には、実効アドレス(すなわち、GR3の値)を番地として持つ主記憶装置の語の値がGR5に格納される。

(3) この問題は、演習問題7-1-1(2)と同じ問題である。プログラム1は、語長が4語になり、主記憶装置とのアクセスが2回になる。プログラム2は、語長が3語になり、主記憶装置とのアクセスが1回になる。

**演習問題7-2-2**

(1) (a) LAD GR1,－1,GR1　(b) LAD GR2,12,GR2　(c) LAD GR3,#00AB,GR3　(d) LAD GR4,#FD03,GR4　(02FD(16)の2の補数はFD03(16))

(2) (a) 指標レジスタにGR0を用いている。　(b) LAD命令はフラグレジスタを変化させないので、JZE命令が正しく働かない。

(3)

```
LAD   GR1,－1,GR1
LAD   GR2,－1,GR2
LAD   GR3,－1,GR3
LAD   GR4,－1,GR4
LAD   GR5,－1,GR5
LAD   GR6,－1,GR6
LAD   GR7,－1,GR7
```

**演習問題7-2-3**

(1)

| 処理内容 | プログラム1 | プログラム2 | プログラム3 | プログラム4 |
|---|---|---|---|---|
| (a) GR3←GR2＋5 | LAD GR3,5,GR2 | LAD GR1,5<br>LD GR3,GR2<br>ADDA GR3,GR1 | LD GR3,GR2<br>ADDA GR3,FIVE<br>FIVE DC 5 | LD GR3,GR2<br>ADDA GR3,=5 |
| (b) GR4←GR1－5 | LAD GR4,－5,GR1 | LAD GR2,5<br>LD GR4,GR1<br>SUBA GR4,GR2 | LD GR4,GR1<br>SUBA GR4,FIVE<br>FIVE DC 5 | LD GR4,GR1<br>SUBA GR4,=5 |
| (別解) | | LAD GR2,－5<br>LD GR4,GR1<br>ADDA GR4,GR2 | LD GR4,GR1<br>ADDA GR4,MFIVE<br>MFIVE DC －5 | LD GR4,GR1<br>ADDA GR4,=－5 |
| (c) GR7←GR6－1 | LAD GR7,－1,GR6 | LAD GR1,1<br>LD GR7,GR6<br>SUBA GR7,GR1 | LD GR7,GR6<br>SUBA GR7,ONE<br>ONE DC 1 | LD GR7,GR6<br>SUBA GR7,=1 |
| (別解) | | LAD GR1,－1<br>LD GR7,GR6<br>ADDA GR7,GR1 | LD GR7,GR6<br>ADDA GR7,MONE<br>MONE DC －1 | LD GR7,GR6<br>ADDA GR7,=－1 |

なお、プログラム2では、他の汎用レジスタ((a)ではGR1,(b)ではGR2)の値を破壊しているので、それを保存しておきたい場合には、次のようにスタックに待避する(7.3.2項参照)。

```
    PUSH   0,GR1
    LAD    GR1,5
    LD     GR3,GR2
    ADDA   GR3,GR1
    POP    GR1
```

(2) GR0が指標レジスタとして使えないことに注意してプログラムを組む。したがって、「LAD GR7,6,GR0」とすることはできない。

```
LAD    GR1,6          LD     GR7,GR0      LD     GR7,GR0
LD     GR7,GR0        ADDA   GR7,SIX      ADDA   GR7,=6
ADDA   GR7,GR1        SIX    DC   6
```

(3) 表7-6の2番目の方法は汎用レジスタを1つ余分に使うが、主記憶装置とのアクセスが発生しない。しかし、3番目と4番目の方法は主記憶装置とのアクセスが発生し、その分、実行速度が低下すると考えられる。ループを使う場合、その内部は何千、何万回も同じ処理が繰り返される可能性があるので、少しでも効率の良い方がよい。

(4) (d)が異なる。(d)を実行するとGR3には$800A_{(16)}$が格納される。これは符号付きで見ると－32758、符号無しで見ると32778となる。

(b)は一見すると－1(カウントダウン)しないように見える。しかし、65535は符号無し数値の中で最大のもので、これを16進数で表現すると$FFFF_{(16)}$となる。これを加算するとオーバーフローするがその結果は－1した値と同じになる。なお、これをチェックするプログラムがAnswerフォルダのchap7フォルダの中のEX723D.caslに格納されている。

**演習問題7-2-4**

(1)
```
ADDA   GR2,GR0
```

(2)
```
ADDA   GR0,GR7
ADDA   GR1,GR7
ADDA   GR2,GR7
ADDA   GR3,GR7
ADDA   GR4,GR7
ADDA   GR5,GR7
ADDA   GR6,GR7
```

(3) シフト命令を使う。ビット操作の方が加算回路を使うより効率が良いと考えられる。

```
SLA  GR3,1
```

**演習問題7-3-1**

(a)
```
PUSH   0,GR1
LD     GR1,GR7
LD     GR7,GR6
LD     GR6,GR1
POP    GR1
```

(b)
```
        ST   GR7,TEMP
        LD   GR7,GR6
        LD   GR6,TEMP

TEMP    DS   1
```

(c)
```
PUSH   0,GR7
PUSH   0,GR6
POP    GR7
POP    GR6
```

**演習問題7-3-2**

(1)
```
PUSH   0,GR1
PUSH   0,GR2
PUSH   0,GR3
PUSH   0,GR4
PUSH   0,GR5
PUSH   0,GR6
PUSH   0,GR7
; 何かの計算
POP    GR7
POP    GR6
POP    GR5
POP    GR4
POP    GR3
POP    GR2
POP    GR1
```

(2)
```
LD  GR7,GR0
PUSH   0,GR7
; 何かの計算
POP    GR0
```

(3)
```
1. PUSH  0,
2. 0,GR0
3. GR0
```

**演習問題7-3-3**

```
PUSH   0,GR4
PUSH   0,GR4
PUSH   0,GR4
; 何かの処理
POP    GR1
POP    GR2
POP    GR3
```

**総合問題7**

**[演習7-1]**

$20=2^2+2^4$, $5=1+2^2$の関係からシフト命令を組み合わせればよい。

```
TE71 START
     LD    GR1,A
     LD    GR2,GR1
     SLA   GR1,2     GR1←4A
     SLA   GR2,4     GR2←16A
     ADDA  GR1,GR2   GR1←4A+16A=20A
     LD    GR3,B
     LD    GR4,GR3
     SLA   GR4,2     GR4←4B
     ADDA  GR3,GR4   GR3←B+4B=5B
     ADDA  GR1,GR3   GR1←20A+5B
     ST    GR1,C
     RET
A    DC    15
B    DC    10
C    DS    1
     END
```

**[演習7-2]**

(方法1)

```
     ADDA GR0,=1
     LAD  GR1,1,GR1
     LAD  GR2,1,GR2
     LAD  GR3,1,GR3
     LAD  GR4,1,GR4
     LAD  GR5,1,GR5
     LAD  GR6,1,GR6
     LAD  GR7,1,GR7
```

(方法2)

```
     ADDA GR0,=1
     ST   GR0,TEMP
     LAD  GR0,1
     ADDA GR1,GR0
     ADDA GR2,GR0
     ADDA GR3,GR0
     ADDA GR4,GR0
     ADDA GR5,GR0
     ADDA GR6,GR0
     ADDA GR7,GR0
     LD   GR0,TEMP

TEMP DS   1
```

(方法3)

```
     ADDA GR0,TEMP
     ST   GR0,TEMP
     LAD  GR0,1
     ADDA GR1,GR0
     ADDA GR2,GR0
     ADDA GR3,GR0
     ADDA GR4,GR0
     ADDA GR5,GR0
     ADDA GR6,GR0
     ADDA GR7,GR0
     LD   GR0,TEMP

TEMP DC   1
```

方法3ではTEMPに最初1を入れておき、その値が不要になった時点でレジスタGR0の退避用に用いている。なお、これを確認するプログラムがTE72.caslに格納されている。参考のために各方法の性能を表に示す。

方法1と方法2の比較

| | プログラムの語数 | 記憶装置とのアクセス |
|---|---|---|
| 方法1 | 16語(命令)+1語(データ) | 1回 |
| 方法2 | 15語(命令)+2語(データ) | 3回 |
| 方法3 | 15語(命令)+1語(データ) | 3回 |

**[演習7-3]**

```
     PUSH 0,GR7
     ADDA GR7,GR6
     ADDA GR7,GR5
     ADDA GR7,GR4
     ADDA GR7,GR3
     ADDA GR7,GR2
     ADDA GR7,GR1
     ADDA GR7,GR0
     ST   GR7,SUM
     POP  GR7
```

なお、確認するためのプログラムがTE73.caslに格納されている。

**演習問題8-1**

(a) 1. GR5　2. JNZ　(b) 1. SKIP　2. JZE LB2　3. JZE　4. JZE　(c) 1. JZE　2. JZE　(d) 1. JPL　2. JZE　3. JUMP (JMIでもよいが多少効率が落ちる。無条件JUMPの方が高速であると考えられる)　(e) 1. JZE　2. JUMP　(f) 1. 1 (あるいは#0001も可) 2. JZE　(g) 1. 3 (あるいは#0003も可)　2. JZE　(h) 1. 7 (あるいは#0007も可)　2. JZE　(i) 1. 1 (あるいは#0001も可)　2. JZE　(j) 1. ADDA　2. CPA GR1,=100　3. JZE

**演習問題8-2-1**

(1)

```
ABS3     START
         LD    GR1,ORGDATA
         JMI   GETABS
SVPROC   ST    GR1,ANSWER
         RET
GETABS   XOR   GR1,=#FFFF
```

```
         ADDA  GR1,=1
         JUMP  SVPROC
ORGDATA  DC    -20
ANSWER   DS    1
         END
```

(2)

(a) 「if A > 0 then N=−A else N=A」とすればよい。

```
EX821A   START
         LD   GR1,A
         JPL  ABSPROC
SAVE     ST   GR1,N
         RET
ABSPROC  XOR  GR1,=#FFFF
         ADDA GR1,=1
         JUMP SAVE
A        DC   16
N        DS   1
         END
```

(b)

```
EX821B   START
         LD   GR1,A
         ADDA GR1,B
         SUBA GR1,C
         JPL  ABSPROC
SAVE     ST   GR1,N
         RET
ABSPROC  XOR  GR1,=#FFFF
         ADDA GR1,=1
         JUMP SAVE
A        DC   16
B        DC   4
C        DC   19
N        DS   1
         END
```

**演習問題8-2-2**

(1) 「CPA GR1,60」の実効アドレスは「60」であるから、これはGR1の値と60番地に格納された値とを比較し、フラグレジスタを変更するという命令になる。60という値と比較したいのならリテラルを用いて「CPA GR1,=60」とすべきである。

(2) 'A' の文字コードは、表4-5から$41_{(16)}$=65であることがわかる。したがって、GR2には65=$41_{(16)}$が格納される。

(3) LAD命令の一般形は、「LAD r,adr[x]」である。adrに記述できるのは、ラベル、10進定数、16進定数、リテラルだけである。したがって、文字定数は記述できないのでCASL IIの構文エラーになる。

**総合問題8**

**[演習8-1]**

```
TE81      START
          LD   GR1,MOJI
          CPA  GR1,=32
          JMI  CONTROL
          JZE  SPACE
          CPA  GR1,=48
          JMI  SEPARATR
          CPA  GR1,=58
          JMI  NUMBER
          CPA  GR1,=65
          JMI  SEPARATR
          CPA  GR1,=91
          JMI  UPPER
          CPA  GR1,=97
          JMI  SEPARATR
          CPA  GR1,=123
          JMI  LOWER
SEPARATR  OUT  ='Separator',=9
          JUMP FINISH
CONTROL   OUT  ='Control Key',=11
          JUMP FINISH
SPACE     OUT  ='Space',=5
          JUMP FINISH
NUMBER    OUT  ='Number',=6
```

```
           JUMP FINISH
UPPER      OUT  =' Upper' ,=5
           JUMP FINISH
LOWER      OUT  =' Lower' ,=5
FINISH     RET
MOJI       DC   '9'
           END
```

**[演習8-2]**

```
TE82       START
           LD   GR1,MOJI
           CPA  GR1,=65
           JMI  SET
           CPA  GR1,=91
           JMI  LOWER
SET        ST   GR1,MOJI
           RET
LOWER      XOR  GR1,=#0020
           JUMP SET
MOJI       DC    'Z'
           END
```

**[演習8-3]**

```
TE83       START
           LD   GR1,A
           ADDL GR1,B
           JOV  OVER
           XOR  GR0,GR0
SET        ST   GR0,OVF
           ST   GR1,C
           RET
OVER       LAD  GR0,1
           XOR  GR1,GR1
           JUMP SET
A          DC   #8000
B          DC   #F000
C          DS   1
OVF        DS   1
           END
```

**演習問題9-2-1**

(1) 1. 2　2. −1　3. GR4　4. JMI（これを確認するプログラムがEX921A.caslに格納されている）
なお、3.は「=9」も可であるが効率が落ちる。

(2)

```
LOOP1A   START
         XOR      GR1,GR1       GR1←0, GR1をカウンタリセット
LOOP     ADDA     GR1,=1        GR1(カウンタ)をカウントアップ
         NOP                  ; ループの処理
         CPA      GR1,=5        GR1:GR4(=5)比較
         JNZ      LOOP          if GR1 ≠ 5 then goto LOOP
         RET
         END

LOOP3A   START
         LAD      GR1,1         GR1←1
LOOP     NOP                  ; ループの処理
         ADDA     GR1,=1        GR1(カウンタ)をカウントアップ
         CPA      GR1,=6        GR1:6 比較
         JNZ      LOOP          if GR1 ≠ 5 then goto LOOP
         RET
         END
```

(3)

```
LOOP3B   START
         LAD      GR1,1         GR1←1
         LAD      GR2,1         GR2←1
         LAD      GR3,6         GR3←6
LOOP     NOP                  ; ループの処理
         ADDA     GR1,GR2       GR1(カウンタ)をカウントアップ
         CPA      GR1,GR3       GR1:6 比較
         JMI      LOOP          if GR1 < 6 then goto LOOP
         RET
         END
```

**演習問題9-2-2**

1. GR1　2. JZE

**演習問題9-2-3**

LAD命令では、フラグレジスタの値を変更しないので、JNZ命令が有効に働かない。ゆえに「LAD GR1,−1,GR1」としたのではだめである。

**演習問題9-3-1(a)**

(1) 汎用レジスタの値をゼロにするには、XOR命令が最も効率が良いと考えられるが、この他に次の命令でも汎用レジスタをゼロクリアできる。表7-3に詳細に記載されている。

```
        LAD  GR1,0
        SUBA GR1,GR1
        LD   GR1,=0
```

(2) 以下に示すようにSUM1Aの方がプログラムの行数は短いが、ループ内での主記憶装置へのアクセスが多くなるので、実行効率が悪くなる。

```
SUM1A   START
        XOR     GR1,GR1      GR1←0, GR1をカウンタリセット
        XOR     GR3,GR3      GR3←0(合計値)
LOOP    ADDA    GR1,=1       GR1(カウンタ)をカウントアップ
        ADDA    GR3,GR1      GR3 = GR3 + GR1(カウンタ)
        CPA     GR1,N        GR1(汎用レジスタ)
        JMI     LOOP         if GR1 < N then goto LOOP
        ST      GR3,S        GR3→S
        RET
S       DS      1
N       DC      10
        END
```

解説：「ADDA GR1,GR4」はレジスタ同士の計算で処理速度が速いし、命令語長も1語である。これに対し、「ADDA GR1,=1」は、値1を主記憶装置から取り出す分だけ処理速度が遅くなり、命令語長も2語になる。「CPA」命令についても同じ事が言える。SUM1Aでは、ループの準備の部分で、2つのロード命令が無くなっているが、そのことよりも何度も繰り返されるループの本体の命令語長や主記憶装置とのアクセス回数が増えていて、トータルの効率がかなり落ちている。

(3)

```
SUM1B   START
        LAD     GR1,1        GR1←1, GR1はカウンタ
        LAD     GR2,1        GR2←1(定数)
        LD      GR4,N        GR4←N, GR4はリミット
        ADDA    GR4,GR2      GR4←N+1(「リミット+1」にする)
        XOR     GR3,GR3      GR3←0(合計値)
LOOP    ADDA    GR3,GR1      GR3 = GR3 + GR1(カウンタ)
        ADDA    GR1,GR2      GR1(カウンタ)をカウントアップ
        CPA     GR1,GR4      GR1:(N+1) 比較
        JMI     LOOP         if GR1 < N+1 then goto LOOP
        ST      GR3,S        GR3→S
        RET
S       DS      1
N       DC      10
        END
```

```
SUM1C       START
            XOR     GR1,GR1      GR1←0, GR1をカウンタリセット
            LD      GR4,N        GR4←N, GR4はリミット
            LAD     GR2,1        GR2←1(定数)
            XOR     GR3,GR3      GR3←0(合計値)
LOOP        CPA     GR1,GR4
            JPL     EXITLOOP
            ADDA    GR3,GR1      GR3 = GR3 + GR1(カウンタ)
            ADDA    GR1,GR2      GR1(カウンタ)をカウントアップ
            JUMP    LOOP
EXITLOOP    ST      GR3,S
            RET
S           DS      1
N           DC      10
            END
```

**演習問題9-3-1(b)**

Nが奇数かどうか判断するには、$0001_{(16)}$と論理積(AND)を取って、最下位ビットを抽出し、それが1であれば奇数、0

であれば偶数と判断できる。最初の命令でNの値はレジスタGR1へロードされているが、この値と0001(16)の論理積を取るとGR1の値がNの値ではなくなってしまう。そこで、GR1の値をいったんGR2にロードして、GR2の値と0001(16)の値の論理積を取る。その結果、1となれば奇数であるから、これを判定する命令はJPLあるいはJNZということになる。もし、Nの値が奇数ならラベルODDVALUEへジャンプする。そこでは、レジスタGR3の値をゼロクリアしてラベルSETVへジャンプする。

1. GR2　2. JPL あるいは JNZ　3. GR3　4. JUMP SETV

**演習問題9-3-2**

以下のプログラムを実行するとNには255が格納される。1～255までの和が32640であり、1～256までの和が32896であるので、N=255が正しいことが分かる。

```
SUM4A     START
          XOR    GR1,GR1       GR1←0, GR1をカウンタリセット
          LAD    GR2,1         GR2←1(定数)
          XOR    GR3,GR3       GR3←0(合計値)
LOOP      ADDA   GR1,GR2       GR1(カウンタ)をカウントアップ
          ADDA   GR3,GR1       GR3 = GR3 + GR1(カウンタ)
          JOV    EXITLOOP      オーバーフローならループを終了
          JUMP   LOOP          LOOPへジャンプ
EXITLOOP  SUBA   GR1,GR2
          ST     GR1,N         GR3→N
          RET
N         DS     1
          END
```

**演習問題9-4-1**

(1)

```
MULTA     START
          LD     GR2,B         GR2←B, B回ループするカウンタ
          LD     GR3,A         GR3←A
          CPA    GR2,GR3       B:A
          JMI    SKIP          B<AならSKIPへジャンプ
          LD     GR1,GR2       BとAを
          LD     GR2,GR3       格納しているレジスタの値を
          LD     GR3,GR1       交換する
SKIP      XOR    GR1,GR1       GR1←0, 解を格納するレジスタ
          LD     GR2,GR2       GR2(ループ回数)とゼロを比較
          JZE    FINISH        ループ回数=0なら答えもゼロ
          LAD    GR4,1         GR4←1, 定数、カウントダウン用
MLTLOOP   ADDA   GR1,GR3       GR1←GR1+GR3 ,積を求める
          SUBA   GR2,GR4       カウンタのカウントダウン
          JPL    MLTLOOP       カウンタが1以上ならMLTLOOPへ
FINISH    ST     GR1,C         GR1(解)→C
          RET
A         DC     0
B         DC     148
C         DS     1
          END
```

プログラムMULTAでは、B≧Aのとき、A,Bの値が入ったレジスタの値を交換しているが、次のように、再ロードする方法も考えられる(MULTA1.casl)。

```
MULTA1    START
          LD     GR2,B         GR2←B, B回ループするカウンタ
          LD     GR3,A         GR3←A
          CPA    GR2,GR3       B:A
          JMI    SKIP          B<AならSKIPへジャンプ
          LD     GR2,A         BとAを
          LD     GR3,B         格納しているレジスタの値を交換する
SKIP      XOR    GR1,GR1       GR1←0, 解を格納するレジスタ
          LD     GR2,GR2       GR2(ループ回数)とゼロを比較
          JZE    FINISH        ループ回数=0なら答えもゼロ
          LAD    GR4,1         GR4←1, 定数、カウントダウン用
MLTLOOP   ADDA   GR1,GR3       GR1←GR1+GR3 ,積を求める
          SUBA   GR2,GR4       カウンタのカウントダウン
          JPL    MLTLOOP       カウンタが1以上ならMLTLOOPへ
FINISH    ST     GR1,C         GR1(解)→C
          RET
A         DC     3
B         DC     148
C         DS     1
          END
```

(2)

```
MULTB    START
         XOR     GR5,GR5       GR5←0(オーバーフローした印)
         LD      GR2,B         GR2←B, B回ループするカウンタ
         LD      GR3,A         GR3←A
         CPA     GR2,GR3       B:A
         JMI     SKIP          B<AならSKIPへジャンプ
         LD      GR1,GR2       BとAを
         LD      GR2,GR3       格納しているレジスタの値を
         LD      GR3,GR1       交換する
SKIP     XOR     GR1,GR1       GR1←0, 解を格納するレジスタ
         LD      GR2,GR2       GR2(ループ回数)とゼロを比較
         JZE     FINISH        ループ回数=0なら答えもゼロ
         LAD     GR4,1         GR4←1, 定数、カウントダウン用
MLTLOOP  ADDA    GR1,GR3       GR1←GR1+GR3 ,積を求める
         JOV     OVERFLOW
         SUBA    GR2,GR4       カウンタのカウントダウン
         JPL     MLTLOOP       カウンタが1以上ならMLTLOOPへ
FINISH   ST      GR1,C         GR1(解)→C
         ST      GR5,OVF       GR5(オーバーフローの印)→OVF
         RET
OVERFLOW LAD     GR1,#7FFF     GR1←正の最大値
         LAD     GR5,1         GR5←1(オーバーフローした印)
         JUMP    FINISH
A        DC      100
B        DC      200
C        DS      1
OVF      DS      1
         END
```

**演習問題9-4-2**

(1)

プログラムCNVTIMEの最初でGR3を0に初期化している。GR3には、分を求めたときの余りが格納されるので初期化の必要性がないと感じるかも知れない。しかし、SECONDが0のときには、「JZE FINISH」命令でFINISHへジャンプする。そのときにGR3が0になっていないと都合が悪いのである。

```
;    秒数(SECOND)を時(HOUR)、分(MINUTE)、秒(SECOND)に変換するプログラム
;
CNVTIME  START
         XOR     GR1,GR1       時
         XOR     GR2,GR2       分
         XOR     GR3,GR3       秒
         LD      GR4,TIME      (ここからどんどん引いていく)
         JZE     FINISH        もしGR3=0ならFINISHへジャンプ
         LAD     GR5,1         GR5←1
; 時間を求める
         LAD     GR6,3600      1時間は3600秒
HLOOP    CPA     GR4,GR6       GR4とGR6(=3600)を比較
         JMI     MSTEP         GR4からGR6(=3600)が引けなくなったらMSTEP
         SUBA    GR4,GR6       GR4←GR4+GR6(=3600)
         ADDA    GR1,GR5       GR1(時間)←GR1+GR5(=1)
         JUMP    HLOOP          HLOOPへジャンプしてループを形成
; 分を求める
MSTEP    LAD     GR6,60        1分間は60秒
MLOOP    CPA     GR4,GR6       GR4とGR6(=60)を比較
         JMI     SSTEP         GR4からGR6(=60)が引けなくなったらSSTEP
         SUBA    GR4,GR6       GR4←GR4+GR6(=3600)
         ADDA    GR2,GR5       GR2(分)←GR2+GR5(=1)
         JUMP    MLOOP          MLOOPへジャンプしてループを形成
; 秒を求める
SSTEP    LD      GR3,GR4       余り(GR4)を秒(GR3)にセット
FINISH   ST      GR1,HOUR      GR1→HOUR
         ST      GR2,MINUTE    GR2→MINUTE
         ST      GR3,SECOND    GR3→SECOND
         RET                    ; プログラムの終了
TIME     DC      12530
HOUR     DS      1
MINUTE   DS      1
SECOND   DS      1
         END
```

(2)

```
DIV5      START
          LAD   GR3,5        GR3←5
          XOR   GR1,GR1      GR1(商)←0
          LD    GR2,A        GR2←A(ここからどんどん引いていく)
          LAD   GR4,1        GR4←1
DIVLOOP   CPA   GR2,GR3      GR2とGR3(=B)を比較
          JMI   FINISH       GR2からGR3(=B)が引けなくなったらFINISH
          SUBA  GR2,GR3      GR2←GR2+GR3(=B)
          ADDA  GR1,GR4      GR1←GR1+GR4(=1)
          JUMP  DIVLOOP      DIVLOOPへジャンプしてループを形成
FINISH    ST    GR1,C        GR1→C
          SLA   GR2,1        余りを2倍にして小数点以下第1桁を求める
          ST    GR2,P        GR2→P
          RET                ; プログラムの終了
A         DC    1253
C         DS    1            商(整数部)
P         DS    1            商(小数点以下第1桁)
          END
```

**総合問題9**

**[演習9-1]**

```
; フィボナッチ数列の第20項目を求める
FIB20     START
          LAD  GR0,0     F0←0
          LAD  GR1,1     F1←1
          LAD  GR3,1     GR3:カウンタ
          LAD  GR4,1     定数
          LAD  GR5,20    定数
LOOP      ADDA GR3,GR4   GR3カウントアップ
          LD   GR2,GR0
          ADDA GR2,GR1   GR2←GR0+GR1
          CPA  GR3,GR5
          JZE  EXITLOOP
          LD   GR0,GR1
          LD   GR1,GR2
          JUMP LOOP
EXITLOOP  ST   GR2,F20
          RET
F20       DS   1
          END
```

**[演習9-2]**

```
LPCHK     START
          LD   GR1,N
          LAD  GR2,1      定数
LOOP      CPA  GR1,GR2
          JZE  FINISH     もしGR1=1ならおしまい
          LD   GR3,GR1    GR1の奇遇判断
          AND  GR3,GR2    (直接GR1とANDを取るとGR1の値が破壊される)
          JZE  EVEN
; 奇数の時の処理
          LD   GR3,GR1
          ADDA GR1,GR3
          ADDA GR1,GR3
          ADDA GR1,GR2     GR1←3GR1+1
          JUMP LOOP
; 偶数の時の処理
EVEN      SRA  GR1,1       GR1←GR1÷2
          JUMP LOOP
FINISH    RET
N         DC   6
          END
```

**[演習9-3]**

IN命令では、データが無くなると、-1の長さを返す。

以下のプログラムは、次々にデータを読み込み、長さが負になったら終了している。

```
FILEPR START
LOOP   IN   TEXT,LEN
       LD   GR1,LEN
       JMI  EXIT
       OUT  TEXT,LEN
       JUMP LOOP
EXIT   RET
TEXT   DS   255
LEN    DS   1
       END
```

**演習問題10-1**

(1) 1. 3　2. 8　3. GR7　4. 3　5. JZE　6. 4　7. SRA　8. 1　9. XOR　10. FFFF　11. 00FF　12. XOR　13.F 000　14. XOR　15. E000　16. AND　17. FF7F　18. AND　19. FEFF　20. OR　21. 0008

注：問題番号1,2,4,6,8は16進表記でも可。

(2) 0,GR3

(3) 3,GR3

**演習問題10-2**

(1)

```
BCOUNT1  START  CSTART
BITON    ADDA GR2,GR4    GR2(1のビット数)をカウントアップ
         JUMP CHECK      CHECKに戻ってループ続行
CSTART   LD   GR1,A
         XOR  GR2,GR2    GR2:1のビット数
         LAD  GR3,16     ループ回数(16回)
         LAD  GR4,1      1(減算用・加算用の定数)
LOOP     SRL  GR1,1      最下位ビットを押し出す
         JOV  BITON      ビットが立っていればBITONへ
CHECK    SUBA GR3,GR4    GR3(カウンタ)をカウントダウン
         JNZ  LOOP       16回回り切ってなければLOOPへジャンプ
         ST   GR2,COUNT  ループ終了、求められたビット数をCOUNTへ
         RET             ;プログラムの終了
A        DC   2263
COUNT    DS   1
         END
```

(2) 1. SLL　2. ADDA GR1,GR2　3. JZE

(3) 1. JOV　2. OR (あるいは、ADDL, ADDAでも動作するが効率が多少落ちると考えられる)

(4) 1. AND　2. 8　3. 1　4. GR4　5. JNZ　6. AND　7. ADDA　8. 0100

**演習問題10-3**

(1)
```
        AND GR4,=#00F0
        SRL GR4,4
```
(2)
```
        AND GR4,=#1FE0
        SRL GR4,5
```
(3)
```
        AND GR4,=#7FFE
        SRL GR4,1
```

なお、(1)～(3)を確認するプログラムがEX10P3.caslに格納されています。

(4) 1. XOR　2. GR1　3. JZE　4. GR5　5. SRL　6. 0F00　7. SUBA　8. 12　9. FINAL

プログラム(DECBIN)は、ループを使わずに各桁ごとに計算しているので多少冗長である。そこで、ループを使ってこのプログラムを簡略化したプログラムが、第11章テーブル操作に載っているので、参考にしてください。

**演習問題 10-4**

1. JOV　2. SLL GR3,1

**演習問題 10-5**

(1) 1. −1　2. SRL GR0,1　3. JZE　4. ADDA GR5,GR4　5. CPA　6. −1

(2) このような問題は、端の部分の処理で間違える可能性が高いから、次のようなデータの組み合わせが考えられる。

```
A   #FFFF   #F000   #0007   #7FFF   #FFFE
N   16      4       3       15      15
```

**演習問題 10-6**

1. SETC　2. XOR　3. JZE　4. SETC

**総合問題10**

**[演習10-1]**

(1) 特定領域を抜き取るマスクデータを作成し、対象データと論理積(AND)を取って、そのビット列だけを抽出する。シフト命令を用いて、抜き取られたビット列の一番低いビット番号のビットが最下位ビットになるようにシフトする。特定領域の対象となるビット数だけループを繰り返し、1ビットずつ右シストで押し出して、そのオーバーフローした数をカウントすれば、それが対象のビット数になっている。

(2) 「XOR GR2,=1」などとして、GR2と1の排他的論理和を取る。こうすれば、もしGR2=1ならばGR2の値は0になり、GR2=0ならばGR2の値は1になる。

(3) 特定領域を含む語がGR1にあるとする。「XOR GR1,=#FFFF」で16ビットすべてのビットを反転させる。対象領域の最下位ビット(問題中の図ならばビット番号6)だけが1のデータを作る。これがGR2にセットされているとする。「ADDA GR1,GR2」を行う。その後、特定領域を抜き取るマスクデータを作成し、対象データと論理積(AND)を取って、そのビットだけを抽出する。

(4) ループを16回繰り返す。この中で、対象となる語を1ビットずつシフトし、オーバーフローをチェックする。フラグを使って連続した状態が12回続いたかどうかチェックする。

(5) 2つの語のビットがすべて一致しているということは、その語が等しいということであるから、このチェックは簡単である。そこで、GR1,GR2,GR3の各汎用レジスタのすべてのビットが等しいかどうかは、次のプログラムで検査できる。

```
        CPA GR1,GR2
        JNZ NO
        CPA GR1,GR3
        JNZ NO
        ; 3つとも等しい

NO      ; 等しくない
```

(6) GR1にABCD(16)が格納されていて、これをBDCA(16)にしてGR2に格納することを考える。これには次のステップが必要になる。(a) GR2をゼロクリアする。(b)GR1をGR3にコピーし、GR3の先頭4ビット(ビット番号(12-15))を抽出する。それを12ビット右に論理シフトしてGR2に加える。(c)GR1をGR3にコピーし、GR3のビット番号(8-11)を抽出する。それを4ビット左に論理シフトしてGR2に加える。(d)GR1をGR3にコピーし、GR3のビット番号(4-7)を抽出する。それをGR2に加える。(e)GR1をGR3にコピーし、GR3のビット番号(0-3)を抽出する。それを8ビット左に論理シフトしてGR2に加える。

(7) GR1にABCD(16)が格納されていて、これをDDDD(16)にしてGR2に格納することを考える。これには次のステップが必要になる。(a) GR2をゼロクリアする。(b)GR1をGR3にコピーし、GR3のビット番号(0-3)を抽出する。それをGR2に加える。(c)GR3を4ビット左に論理シフトしてGR2に加える。(d)GR3を8ビット左に論理シフトしてGR2に加える。(e)GR3を12ビット左に論理シフトしてGR2に加える。

**[演習10-2]**

(1)

```
SUMPM1  START
        XOR     GR1,GR1         GR1←0, GR1をカウンタリセット
        LD      GR4,N           GR4←N, GR4はリミット
        LAD     GR2,1           GR2←1(定数)
        XOR     GR3,GR3         GR3←0(合計値)
        LAD     GR5,0           フラグ:1なら+, 0なら−
LOOP    ADDA    GR1,GR2         GR1(カウンタ)をカウントアップ
        XOR     GR5,GR2         フラグを反転
        JPL     PLUS            +ならPLUSへ
        SUBA    GR3,GR1         GR3 = GR3 − GR1(カウンタ)
        JUMP    SKIP
PLUS    ADDA    GR3,GR1         GR3 = GR3 + GR1(カウンタ)
SKIP    CPA     GR1,GR4         GR1(汎用レジスタ)
        JMI     LOOP            if GR1 < N then goto LOOP
        ST      GR3,S           GR3→S
        RET
S       DS      1
N       DC      100
        END
```

(2)

```
SUMPM2  START
        XOR     GR1,GR1         GR1←0, GR1をカウンタリセット
        LAD     GR4,9           GR4←9, GR4はリミット(ループ回数)
        LAD     GR2,1           GR2←1(定数)
        XOR     GR3,GR3         GR3←0(合計値)
        LAD     GR5,0           フラグ:1なら+, 0なら−
        LAD     GR6,1           GR6←1,加える値
```

```
LOOP     ADDA    GR1,GR2      GR1(カウンタ)をカウントアップ
         XOR     GR5,GR2      フラグを反転
         JPL     PLUS         +ならPLUSへ
         SUBA    GR3,GR6      GR3 = GR3 - GR1(カウンタ)
         JUMP    SKIP
PLUS     ADDA    GR3,GR6      GR3 = GR3 + GR1(カウンタ)
SKIP     SLA     GR6,1        GR6 = 2 x GR6
         CPA     GR1,GR4      GR1(汎用レジスタ)
         JMI     LOOP         if GR1 < N then goto LOOP
         ST      GR3,S        GR3→S
         RET
S        DS      1
         END
```

**[演習10-3]**

次に示すプログラムのように、処理内容のAをビットの1、Bをビットの0に見立てて、データC5CB(16)を作成する。このデータをシフト命令で左側(上位ビット)から押し出し、オーバーフローしていればAの処理を行い、オーバーフローしていなければBの処理を行う。以下のプログラムでは、Aの処理では'A'と表示し、Bの処理では'B'と表示している。

```
ABCHECK  START
         LD    GR1,COM
         LAD   GR2,1
LOOP     SLL   GR1,1
         JOV   A
B        OUT   ='B',=1
         JUMP  EXITCHK
A        OUT   ='A',=1
EXITCHK  SLL   GR2,1
         JOV   FINISH
         JUMP  LOOP
FINISH   RET
COM      DC    #C5CB
         END
```

**[演習10-4]**

```
BCOUNTE  START
         LD    GR1,A
         XOR   GR2,GR2      GR2:1のビット数
         LAD   GR3,16       ループ回数(16回)
         LAD   GR4,1        1(減算用・加算用の定数)
LOOP     LD    GR1,GR1      GR1の値でフラグレジスタをセット
         JZE   FINISH       もしGR1がゼロならば終了
         SRL   GR1,1        最下位ビットを押し出す
         JOV   BITON        ビットが立っていればBITONへ
CHECK    SUBA  GR3,GR4      GR3(カウンタ)をカウントダウン
         JNZ   LOOP         16回回り切ってなければLOOPへジャンプ
FINISH   ST    GR2,COUNT    ループ終了、求められたビット数をCOUNTへ
         RET                ;プログラムの終了
BITON    ADDA  GR2,GR4      GR2(1のビット数)をカウントアップ
         JUMP  CHECK        CHECKに戻ってループ続行
A        DC    2263
COUNT    DS    1
         END
```

**演習問題11-2-1**

テーブル中の各要素を参照する場合には、多くの場合、「テーブルの先頭アドレス+指標レジスタの値」という実効アドレスを生成し、このアドレスによってテ--ブル中のデータをアクセスする。そのため、最初の要素をアクセスするには、指標レジスタの値は0にならざるを得ないのである。

**演習問題11-2-2**

(1) 1. 25　2. JMI　3. GR1,SEQ,GR1
(注：1.を24としないこと。SEQの25番目がセットされなくなる)
(2) 1. SUBA GR2,GR3　2. GR2

**演習問題11-3**

1. XOR　2. LD　GR5,HDATA,GR1　3. JZE　4. GR5　5. GR3

**演習問題11-4-1**

1. JZE　FOUND　2. LD GR5,GR1　3. SETV

**演習問題11-4-2**

COUNTTプログラムでは、テーブルMTABLEのデータをループのたびごとにレジスタGR5にロードして、そのGR5のデータと定数1が格納されているレジスタGR3の論理積を取って、奇遇の判断をしている。これでは、効率が悪いと考えて、レジスタGR3とテーブルMTABLEの値を直接的に論理積を取っているのが問題文に示された命令「AND GR3,MTABLE,GR1」である。

一見うまく行くように見えるが、実は、これではうまく動作しない。なぜなら、定数1を格納したレジスタGR3の値が、対象となるMTABLEの要素が偶数のときに、上記のAND命令の実行によって0になってしまうからである。

**演習問題11-4-3**

(1) 「LD GR2,MTABLE」によって生成される実効アドレスは「MTABLE」そのものであり、これはテーブルMTABLEの先頭アドレスである。その値がGR2にロードされる。

(2) 1. JMI　2. JZE

(3) 答えは求められる。しかし、無駄な比較が最初に1回行われる。つまり、最初のデータをとりあえずの最大値としているのに、その値と(同じ)最初の値を比較することになる。

**演習問題11-5-1**

(1) 1. JMI　2. GR3　3. XOR　GR5,GR5　4. 9　5. ADDA　6. FFFF　7. SETDIGIT　8. 1,16,256,4096

(2) '7'の文字コードは、表4-5の文字コード表によると、$37_{(16)}$である。これに10を加えると$41_{(16)}$となり、これは'A'の文字コードになる。また、$37_{(16)}$に15を加えると$46_{(16)}$となり、これは'F'の文字コードになる。

**演習問題11-5-2**

(1) 1. JPL　2. 4　3. 'A'

(2) うまくいかない。SLAでは、最上位ビットを符号と見なすため、この場合には、右側からシフトしているので、必ず0になる。

**総合問題11**

**[演習11-1]**

```
TOCHAR    START
          XOR  GR1,GR1      テーブルMESSAGEのインデックス
          XOR  GR2,GR2      文字数
          LAD  GR3,1        1:定数
LOOP      LD   GR4,MESSAGE,GR1
          JZE  FINISH       0ならおしまい
          CPL  GR4,='a'
          JMI  NEXT         'a'より小さければ次へ
          CPL  GR4,='z'
          JPL  NEXT         'z'より大きければ次へ
          AND  GR4,=#FFDF 小文字→大文字
NEXT      ST   GR4,OUTAREA,GR1
          ADDA GR1,GR3      インデックス+1
          ADDA GR2,GR3      文字数+1
          JUMP LOOP
FINISH    ST   GR2,OUTLEN
          OUT  OUTAREA,OUTLEN
          RET
MESSAGE   DC    'This is a long long message'
          DC   0
OUTAREA   DS   255
OUTLEN    DS   1
          END
```

**[演習11-2]**

```
BINSRCH   START
          LD   GR0,A        GR0←A
          LAD  GR1,1        1:定数
          XOR  GR2,GR2      GR2←0, 先頭
          LD   GR3,N        GR3←N, 個数
          SUBA GR3,GR1      GR3←N-1, 最後の要素
LOOP      LD   GR4,GR3
          SUBA GR4,GR2      GR4←(最後-先頭)
```

```
        SRA   GR4,1        GR4←(最後-先頭) / 2
        LD    GR5,GR2
        ADDA GR5,GR4      GR5:チェックする位置
        LD    GR6,WTABLE,GR5
        CPA   GR6,GR0
        JZE   FOUND        見つかったか
        CPA   GR2,GR3      インデックスのチェック
        JZE   NOTFOUND     見つからなかったか
        CPA   GR6,GR0      右か左のどちらかにある
        JPL   SMALL
LARGE   ADDA GR5,GR1      左にある
        LD    GR2,GR5
        JUMP LOOP
SMALL   SUBA GR5,GR1      右にある
        LD    GR3,GR5
        JUMP LOOP
FOUND   ST    GR5,FPOS
        RET
NOTFOUND LAD  GR5,-1
        RET
FPOS    DS    1
WTABLE  DC     -1000,-233,-58,-34,-12,0,12,122,125,456,3021,4404,9800
N       DC     13
A       DC     -58
        END
```

**[演習11-3]**

```
KEYSUM   START
         LAD   GR0,1         1:定数
;  出力領域のクリア
         LD    GR1,OUTLEN
         LD    GR2,='  '
CLEARLP  SUBA GR1,GR0
         JMI   STARTPGM
         ST    GR2,OUTAREA,GR1
         JUMP CLEARLP
STARTPGM LD    GR7,OUTLEN    出力領域の長さ
         SUBA GR7,GR0
         IN    N1,N1LEN      入力データの取得
         IN    N2,N2LEN
         XOR   GR6,GR6       桁上がりをゼロにセット
         LD    GR1,N1LEN
         LD    GR2,N2LEN
         SUBA GR1,GR0       インデックスなので
         SUBA GR2,GR0       1つ減じる
LOOP     LD    GR3,='0'      数字のゼロをセット
         LD    GR4,='0'      数字のゼロをセット
         LD    GR1,GR1
         JMI   FIN1          N1の領域が無なくなったか
         LD    GR3,N1,GR1    N1の桁をセット
         JUMP NEXTCHK
FIN1     LD    GR2,GR2       N2の桁が無くなったか
         JMI   FINISH        両方の桁が無くなったら終了
NEXTCHK  LD    GR2,GR2
         JMI   FIN2          N2の桁が無くなったか
         LD    GR4,N2,GR2    N2の桁をセット
FIN2     SUBA GR3,='0'      数字を数値に変換
         SUBA GR4,='0'      数字を数値に変換
         ADDA GR3,GR6       前の桁上がりを加える
         XOR   GR6,GR6       次のために桁上がりをクリアする
         ADDA GR3,GR4       桁を加える
         CPA   GR3,=10       10以下か
         JMI   SKIP
         LAD   GR6,1         桁上がりあり!
         SUBA GR3,=10
SKIP     ADDA GR3,='0'      数値を数字に戻す
         ST    GR3,OUTAREA,GR7 出力領域に転送
         SUBA GR1,GR0
         SUBA GR2,GR0
         SUBA GR7,GR0
         JUMP LOOP
FINISH   LD    GR6,GR6       桁上がりがあるか
         JZE   NO            なければNOへ
```

```
          ADDA GR6,=' 0'       桁上がりを文字へ
          ST   GR6,OUTAREA,GR7 出力領域へ転送
NO        LD   GR1,N1LEN       メッセージを付けて
          ADDA GR1,MESLEN      データを表示する
          ST   GR1,N1LEN
          OUT  N1MES,N1LEN
          LD   GR2,N2LEN
          ADDA GR2,MESLEN
          ST   GR2,N2LEN
          OUT  N2MES,N2LEN
          LD   GR3,OUTLEN
          ADDA GR3,MESLEN
          ST   GR3,OUTLEN
          OUT  OUTMES,OUTLEN
          RET
MESLEN    DC   8
N1MES     DC   'DATA1 =  '
N1        DS   10
N1LEN     DS   1
N2MES     DC   'DATA2 =  '
N2        DS   10
N2LEN     DS   1
OUTMES    DC   'ANSWER=  '
OUTAREA   DS   255
OUTLEN    DC   10
          END
```

**[演習11-4]**

BINDECR.caslに答えがある。

ここには、ゼロ抑制する部分だけを載せる。

;先頭の空白を削除する処理

```
          LAD  GR2,5
          ST   GR2,DLEN
          XOR  GR1,GR1          DIGITのインデックス
          LD   GR0,DIGIT,GR1    先頭の文字を取り出す
          CPA  GR0,=' -'        負記号か
          JNZ  NOTMINUS
          LAD  GR2,6
          ST   GR2,DLEN
          ADDA GR1,=1           インデックスを次に進める
          LD   GR0,DIGIT,GR1    次の文字を取る
NOTMINUS  CPA  GR0,=' 0'        文字のゼロか
          JNZ  PRINT            ゼロでなければ先行するゼロはない
          LD   GR7,GR1          GR7:詰めるゼロの先頭インデックス
LOOP      ADDA GR1,=1
          LD   GR0,DIGIT,GR1    次の文字を取る
          CPA  GR0,=' 0'
          JNZ  FOUNDDG          0以外の数字発見
          CPA  GR1,=5
          JZE  LOOPFIN          全部0か
          JUMP LOOP
FOUNDDG   ST   GR0,DIGIT,GR7    数字発見,詰める
          ADDA GR1,=1
          ADDA GR7,=1
          LD   GR0,DIGIT,GR1
          CPA  GR1,DLEN
          JMI   FOUNDDG
          ST    GR7,OUTLEN      長さなので1つ余分にする
          JUMP  PRINT
LOOPFIN   ST    GR0,DIGIT
          LAD   GR1,1
          ST    GR1,OUTLEN
PRINT     OUT  DIGIT,OUTLEN     編集領域を出力
          RET                 ; プログラムの終了
A         DC   22315              変換する数値
WTABLE    DC   10,100,1000,10000  10進数の重み
DIGIT     DS   6                  編集領域
OUTLEN    DS   1                  編集領域の長さ
DLEN      DS   1                数字の長さ(-のとき6,正のとき5)
          END
```

**演習問題12-1-1**

(1) プログラム中に記述されているのに一度も実行されないプログラムがあっても、その部分は他の部分と同様にアセンブラによって機械語に変換される。そして、それがプログラムの一部として毎回、主記憶装置にロードされてしまう。実行そのものには影響を与えないが、プログラムサイズが無意味に大きくなり、記憶効率が悪くなると言える。

(2) 条件判断を用いてある条件の時だけサブルーチンを実行させるようにすると、問題で指摘されているプログラムを作ることができる。

**演習問題12-1-2**

(1)

| 値 | 番地 | 内容 |
|---|---|---|
| 7 | $752D_{(16)}$ | サブルーチンS2の戻り番地　　SP [752D] |
| 3 | $752E_{(16)}$ | PUSH 3によって積まれた値 |
| 2 | $752F_{(16)}$ | サブルーチンS1の戻り番地 |
| ???? | $7530_{(16)}$ | この領域は実際には使われないので何があるか不明（OSへの戻り番地の可能性もある） |

(2) サブルーチンの戻り番地はスタック上に置かれる。したがって、スタックが利用できなくなるほど多くの戻り番地をスタックに積み上げるほどCALL命令を実行すれば、利用不可能になる。なお、スタックは、使用中のプログラムの領域に達するか0番地に達すると利用不可能になる。しかし、COMET IIでは、このチェックを自分のプログラムの中でかけることは不可能である(が、本書に付属のCASLDVではチェックしている)。情報処理技術者試験の問題の中では、十分な量のスタック領域があるものと仮定されている。

**演習問題12-1-3**

(1) 「CALL SUB1」で呼び出されたサブルーチンSUB1は、サブルーチンとして実行される。その後、「JUMP SUB1」でラベルSUB1のある命令に制御が移るが、その中にあるRET命令によってオペレーティングシステムに制御が戻る。つまり、プログラムの実行が終了する。その意味で、「JUMP SUB1」の下のRET命令は決して実行されることのない命令になる。

(2) CASLDVにかけるとサブルーチンの中でスタックのPUSHとPOPの対応が合っていないという実行時エラーが報告される。チェック機能をはずす(メニューのオプション→実行時オプションの指定→「サブルーチン内のスタックの使用状況をチェックする」のチェックをはずす)と「PUSH 0,GR1」でスタックにPUSHしたGR1の値をサブルーチンの戻り番地としてRETしてしまう。

**演習問題12-1-4**

演習問題12-1-3にも示したとおり、一つのサブルーチンの中でPUSHとPOPの個数が合っていないと、RET命令で正しくサブルーチンの戻り番地に戻れなくなる。

**演習問題12-2-1**

1. PUSH 0,GR3　2. POP GR3

**演習問題12-2-2**

サブルーチンの中で汎用レジスタの値を変更しないようにするには、スタックを用いる。

```
        PUSH  0,GR3
        PUSH  0,GR2
        PUSH  0,GR1
        LD    GR1,GR0
        PUSH  0,GR1         GR0を退避

        サブルーチン MULTS2 の実行

        POP   GR0
        POP   GR1
        POP   GR2
        POP   GR3
```

**演習問題12-3**

1. JPL SHIFTLP　2. JMI NEXTBIT

**演習問題12-4**

```
; 再帰呼び出しでNの階乗を求めるプログラム
; ただし、Nの値は7までしかできない
;
; メインルーチン
RFACT     START
          LD     GR1,N
          CALL   FACT
          ST     GR1,F
          RET
N         DC     7
```

```
F          DS     1
;
; 1からNまでの階乗を求めるサブルーチン
; 入力 GR1:N, 出力 GR1:階乗
FACT       CPA    GR1,=0       もし、N(GR1)がゼロかどうか
           JNZ    CAL          ゼロでなければCALへ
           LAD    GR1,1        GR1←1
           RET                 ;ゼロならば、返却値は1(FACT(0)=1)で終了
CAL        PUSH   0,GR1        N の 値をプッシュして保存する
           SUBA   GR1,=1       GR1 ← N - 1 ( GR1-1 )
           CALL   FACT         FACTを呼ぶ。すなわち、 GR1 ← FACT( N-1 )を計算する。
           POP    GR2          N の 値を復帰する
           ;GR3 ← GR1 x GR2 = S( N-1 ) x N, GR2はゼロではない
           XOR    GR3,GR3      GR3←0
MULTLOOP   ADDA   GR3,GR1      GR3←GR3+GR1
           SUBA   GR2,=1       GR2←GR2-1,カウントダウン
           JNZ    MULTLOOP
           LD     GR1,GR3
           RET                 ; GR1( S(N-1) x N )を返却値としてサブルーチンを終了
           END
```

**総合問題12**

**[演習1]**

1. LAD　2. LD　GR6,0,GR1　3. LD　4. ADDA　GR1,GR3

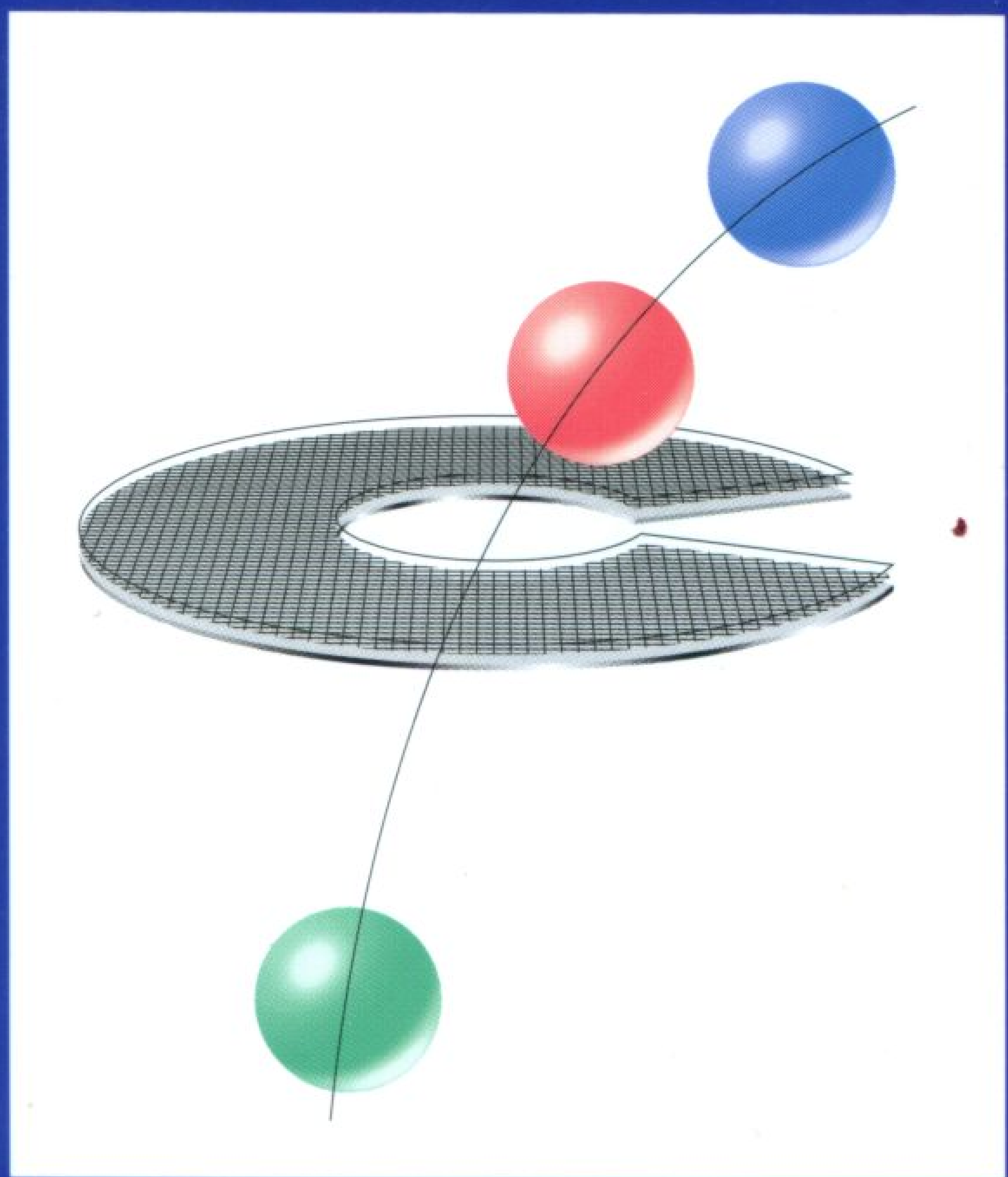
**[演習2]**

1. LD　GR5,='¥'　2. SUBA GR5,GR6　3. ADDA GR4,GR7　4. LD　GR6,=','

ISBN4-88647-607-4
C3055 ¥2500E
定価 (本体2,500円+税)
発行 電子開発学園出版局
発売 株式会社SCC

情報処理基礎講座

# アセンブラ入門 CASLⅡ



電子開発学園出版局